Competitive Security Assessment

Magma_Update

Sep 8th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

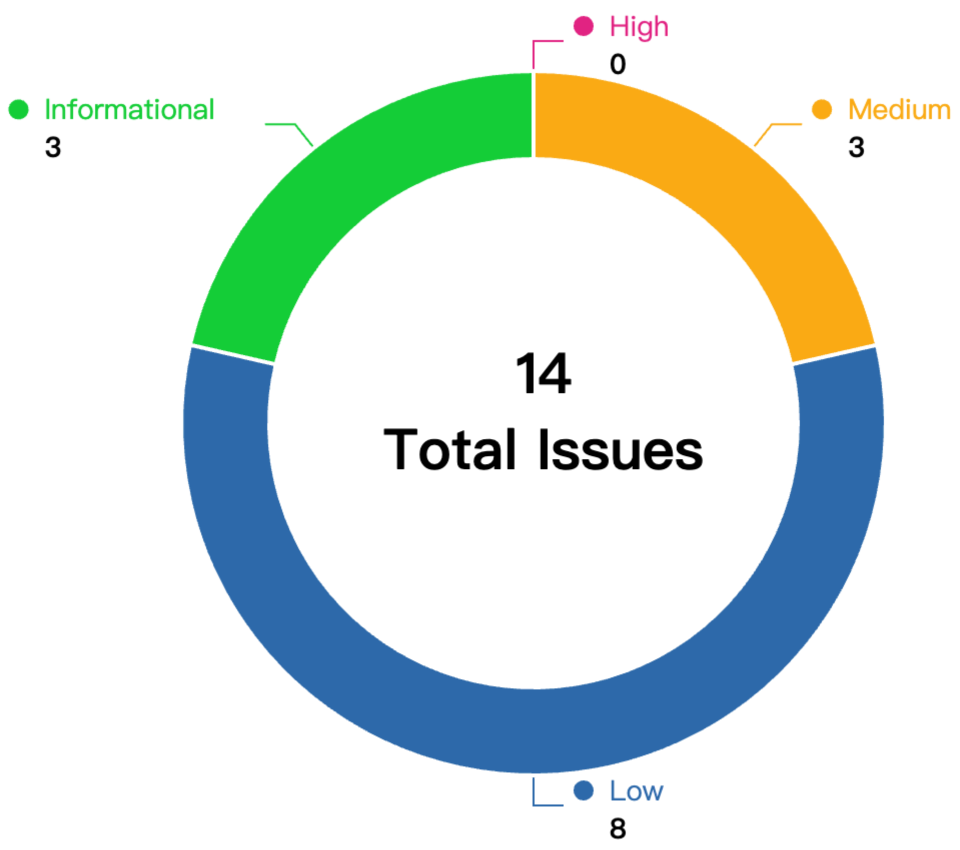
Overview

Project Name	Magma_Update
Language	solidity
Codebase	• https://github.com/magma-fi/WEN-Contracts/compare/997d1a13a37b2da0938f6d89946c1365bbececd0...f30cf0f196ffe424696e8cf3f341b41e3f4e5a9e • audit version-f30cf0f196ffe424696e8cf3f341b41e3f4e5a9e • final version-1095847b976534d90c093822b9fbf4d237ef8817

Audit Scope

File	SHA256 Hash
contracts/TroveManager.sol	36f7c5c268d81b9b31318f4fa8e712cf054bff095dd829609ea0f6996dd1a39d
contracts/BorrowerOperations.sol	d39317f7ab3d4e263e132d4d542c4a660a9afbe0dd98d8401c463a6492d62540
contracts/Dependencies/LiquityBase.sol	db29c52a48d3c3fa1a1023fb56d020a85df7c34eced39fb41575b59b014b44eb
contracts/Dependencies/CheckContract.sol	b61f9160d8a2a358faa898ab439c15a6969302f81d732b3a4a136b51be877e41
contracts/Dependencies/SysConfig.sol	14d33de1c608689a31791e27e9dec29575c80602235f217f870c3f68a39262c0
contracts/CollTokenPriceFeed.sol	f924272a751bbcbdc0a8fa0c9b4b2f1c12f686f22596b24e5b55ee95bc7b4798

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
MAG-1	The function burnLUSD will burn twice as many tokens as expected	Logical	Medium	Fixed	***
MAG-2	PriceFeed oracle can return stale price	DOS	Medium	Fixed	***
MAG-3	ERC20 with small decimal is not supported	Logical	Medium	Fixed	***
MAG-4	Use safeTransferFrom instead of Transfer	Logical	Low	Fixed	***
MAG-5	The `setAddresses` function does not check for duplicate `_collToken`	Logical	Low	Fixed	***
MAG-6	The `onReceive` function does not check if the received token is the same with the `collToken`	Logical	Low	Fixed	***

MAG-7	Missing Check the `msg.value` if the `_collToken` is not native token	Logical	Low	Fixed	***
MAG-8	Incompatible With the Deflationary or Fee-on-Transfer Tokens in Function	Logical	Low	Fixed	***
MAG-9	If a trove is opened by a contract, it may not be closed any more	Logical	Low	Fixed	***
MAG-10	Hardcoded TIMEOUT is not appropriate for all tokens	Logical	Low	Fixed	***
MAG-11	Approve may revert if the `collToken` is USDT	Logical	Low	Fixed	***
MAG-12	Unsafe casting	Integer Overflow and Underflow	Informational	Fixed	***
MAG-13	The error message is incorrect	Code Style	Informational	Fixed	***
MAG-14	Redundant code	Code Style	Informational	Fixed	***

MAG-1:The function burnLUSD will burn twice as many tokens as expected

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/contracts/BorrowerOperations.sol#L452-L455

```
452: function burnLUSD(uint _amount) external override {
453:         lusdToken.transferFrom(msg.sender, address(this), _amount);
454:         lusdToken.burn(msg.sender, _amount);
455:     }
```

Description

***: In `BorrowerOperations` contract, the `burnLUSD` function is used for users to burn their LUSD tokens. It will first transfer the LUSD from the user to the `BorrowerOperations` contract:

```
lusdToken.transferFrom(msg.sender, address(this), _amount);
```

and then it will call `lusdToken.burn` with the first param as `msg.sender`, which means the user will burn the same amount of tokens:

```
lusdToken.burn(msg.sender, _amount);
```

As a result, the user loses two parts of the token, the first is transferred to the contract and the second is burned directly. Since there is no withdraw function in `BorrowerOperations` contract, the tokens transferred to the `BorrowerOperations` contract will be locked forever.

Recommendation

***: Consider following fix:

```
function burnLUSD(uint _amount) external override {
        lusdToken.transferFrom(msg.sender, address(this), _amount);
        lusdToken.burn(address(this), _amount);
    }
```

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-2:PriceFeed oracle can return stale price

Category	Severity	Client Response	Contributor
DOS	Medium	Fixed	***

Code Reference

- code/contracts/CollTokenPriceFeed.sol#L83-L104
- code/contracts/CollTokenPriceFeed.sol#L116-L131

```
83: function fetchPrice(address _collToken) external override returns (uint) {
84:         // Get current and previous price data from Chainlink
85:         ChainlinkResponse memory chainlinkResponse = _getCurrentChainlinkResponse(collTokenPrice
Feed[_collToken].priceAggregator);
86:         ChainlinkResponse memory prevChainlinkResponse = _getPrevChainlinkResponse(chainlinkResp
onse.roundId, chainlinkResponse.decimals, collTokenPriceFeed[_collToken].priceAggregator);
87:
88:         if (collTokenPriceFeed[_collToken].status == Status.chainlinkWorking) {
89:             if (_chainlinkIsBroken(chainlinkResponse, prevChainlinkResponse)) {
90:                 _changeStatus(_collToken, Status.oraclesUntrusted);
91:                 return collTokenPriceFeed[_collToken].lastGoodPrice;
92:             }
93:
94:             // If Chainlink is working, return Chainlink current price (no status change)
95:             return _storeChainlinkPrice(_collToken, chainlinkResponse);
96:         }
97:         if (collTokenPriceFeed[_collToken].status == Status.oraclesUntrusted) {
98:             if (!_chainlinkIsBroken(chainlinkResponse, prevChainlinkResponse)) {
99:                 _changeStatus(_collToken, Status.chainlinkWorking);
100:                 return _storeChainlinkPrice(_collToken, chainlinkResponse);
101:             }
102:         }
```

```
103:         return collTokenPriceFeed[_collToken].lastGoodPrice;
104:     }
```

```
116: function _chainlinkIsBroken(ChainlinkResponse memory _currentResponse, ChainlinkResponse memory
_prevResponse) internal view returns (bool) {
117:         return _badChainlinkResponse(_currentResponse) || _badChainlinkResponse(_prevResponse);
118:     }
119:
120:     function _badChainlinkResponse(ChainlinkResponse memory _response) internal view returns (b
ool) {
121:          // Check for response call reverted
122:         if (!_response.success) {return true;}
123:         // Check for an invalid roundId that is 0
124:         if (_response.roundId == 0) {return true;}
125:         // Check for an invalid timeStamp that is 0, or in the future
126:         if (_response.timestamp == 0 || _response.timestamp > block.timestamp) {return true;}
127:         // Check for non-positive price
128:         if (_response.answer <= 0) {return true;}
129:
130:         return false;
131:     }
```

- code/contracts/UniIOTXPrice.sol#L30-L46

```
function getRoundData(uint80 _roundId) public view override returns (
                                                                            uint80 roundId,
                                                                            int256 answer,
                                                                            uint256 startedAt,
                                                                            uint256 updatedAt,
                                                                            uint80 answeredInRound) {
            roundId = _roundId;
            uint exchangeRatio = iotxStaking.exchangeRatio();
            (, int currentIOTXPrice,,,) = iotxPriceOracle.latestRoundData();
            answer = int(exchangeRatio) * currentIOTXPrice / 1e18;
            startedAt = block.timestamp;
            updatedAt = block.timestamp;
            answeredInRound = roundId;

            require(answer != 0, "price is 0");
            require((answer + currentIOTXPrice)/currentIOTXPrice == 2, " invalid price");
        }
```

Description

***: In contract `UniIOTXPrice`, the function `getRoundData` will use `AggregatorV3Interface.latestRoundData()` to fetch latest price:

```
(, int currentIOTXPrice,,,) = iotxPriceOracle.latestRoundData();
```

The issue here is that there is no check for the last updated time for the price. So we would not know if the price returned exceeded the timeout. It may return an expired price and incur unexpected side effects.
The same issue exists in `CollTokenPriceFeed`. The `fetchPrice` function will call `_chainlinkIsBroken` function to check whether the price is valid:

```
function _chainlinkIsBroken(ChainlinkResponse memory _currentResponse, ChainlinkResponse memory _prevResponse) internal view returns (bool) {
        return _badChainlinkResponse(_currentResponse) || _badChainlinkResponse(_prevResponse);
    }

    function _badChainlinkResponse(ChainlinkResponse memory _response) internal view returns (bool) {
         // Check for response call reverted
        if (!_response.success) {return true;}
        // Check for an invalid roundId that is 0
        if (_response.roundId == 0) {return true;}
        // Check for an invalid timeStamp that is 0, or in the future
        if (_response.timestamp == 0 || _response.timestamp > block.timestamp) {return true;}
        // Check for non-positive price
        if (_response.answer <= 0) {return true;}

        return false;
    }
```

However, the `_chainlinkIsBroken` does not check whether the price returned exceeded the timeout.

Recommendation

***: In `UniIOTXPrice`, consider adding a check to see when the price was last updated and revert if the price is older than a certain time period:

```
(
                /* uint80 roundID */,
                int256 currentIOTXPrice,
                /*uint startedAt*/,
                uint updatedAt,
                /*uint80 answeredInRound*/
            ) = iotxPriceOracle.latestRoundData();
            require(currentIOTXPrice > 0, "Error: Invalid price");
            require(updatedAt > block.timestamp - MAX_TIME_DELAY, "Error: Invalid updated time");
```

In `CollTokenPriceFeed`, consider following fix:

```
function fetchPrice(address _collToken) external override returns (uint) {
        // Get current and previous price data from Chainlink
        ChainlinkResponse memory chainlinkResponse = _getCurrentChainlinkResponse(collTokenPriceFe
ed[_collToken].priceAggregator);
        ChainlinkResponse memory prevChainlinkResponse = _getPrevChainlinkResponse(chainlinkRespon
se.roundId, chainlinkResponse.decimals, collTokenPriceFeed[_collToken].priceAggregator);
        if (_chainlinkIsFrozen(chainlinkResponse)) {
                return collTokenPriceFeed[_collToken].lastGoodPrice;
        }
        ...
        ...
```

Client Response

client response : Fixed. Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-3:ERC20 with small decimal is not supported

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/contracts/BorrowerOperations.sol#L169-L215

```
169: function openTrove(address _collToken, uint _collAmount, uint _maxFeePercentage, uint _LUSDAmount, address _upperHint, address _lowerHint) public payable {
170:         sysConfig.checkCollToken(_collToken);
171:
172:         if (isNativeToken(_collToken)) {
173:             _collAmount = msg.value;
174:         } else {
175:             IERC20(_collToken).transferFrom(msg.sender, address(this), _collAmount);
176:         }
177:         ITroveManager localTroveManager = _getTroveManager(_collToken);
178:         ContractsCache memory contractsCache = ContractsCache(localTroveManager, IActivePool(_getActivePool(_collToken)), lusdToken);
179:         LocalVariables_openTrove memory vars;
180:
181:         vars.price = sysConfig.fetchPrice(_collToken);
182:
183:         bool isRecoveryMode;
184:         if (isNativeToken(_collToken)) {
185:             isRecoveryMode = _checkRecoveryMode(vars.price);
186:         } else {
187:             isRecoveryMode = sysConfig.checkRecoveryMode(_collToken, vars.price);
188:         }
```

```
189:
190:         _requireValidMaxFeePercentage(_maxFeePercentage, isRecoveryMode);
191:         _requireTroveisNotActive(contractsCache.troveManager, msg.sender);
192:
193:         vars.LUSDFee;
194:         vars.netDebt = _LUSDAmount;
195:
196:         if (!isRecoveryMode) {
197:             vars.LUSDFee = _triggerBorrowingFee(contractsCache.troveManager, contractsCache.lusdToken, _LUSDAmount, _maxFeePercentage);
198:             vars.netDebt = vars.netDebt.add(vars.LUSDFee);
199:         }
200:         _requireAtLeastMinNetDebt(vars.netDebt);
201:
202:         // ICR is based on the composite debt, i.e. the requested LUSD amount + LUSD borrowing fee + LUSD gas comp.
203:         vars.compositeDebt = _getCompositeDebt(_collToken, vars.netDebt);
204:         assert(vars.compositeDebt > 0);
205:
206:         vars.ICR = LiquityMath._computeCR(_collAmount, vars.compositeDebt, vars.price);
207:         vars.NICR = LiquityMath._computeNominalCR(_collAmount, vars.compositeDebt);
208:
        if (isRecoveryMode) {
            _requireICRisAboveCCR(_collToken, vars.ICR);
        } else {
            _requireICRisAboveMCR(_collToken, vars.ICR);
            uint newTCR = _getNewTCRFromTroveChange(_collToken, _collAmount, true, vars.compositeDebt, true, vars.price);  // bools: coll increase, debt increase
            _requireNewTCRisAboveCCR(_collToken, newTCR);
        }
```

- code/contracts/CollTokenStabilityPool.sol#L681-L714

```
/* Calculates the ETH gain earned by the deposit since its last snapshots were taken.
    * Given by the formula:  E = d0 * (S - S(0))/P(0)
    * where S(0) and P(0) are the depositor's snapshots of the sum S and product P, respectively.
    * d0 is the last recorded deposit value.
    */
    function getDepositorETHGain(address _depositor) public view override returns (uint) {
        uint initialDeposit = deposits[_depositor].initialValue;

        if (initialDeposit == 0) { return 0; }

        Snapshots memory snapshots = depositSnapshots[_depositor];

        uint ETHGain = _getETHGainFromSnapshots(initialDeposit, snapshots);
        return ETHGain;
    }

    function _getETHGainFromSnapshots(uint initialDeposit, Snapshots memory snapshots) internal view returns (uint) {
        /*
        * Grab the sum 'S' from the epoch at which the stake was made. The ETH gain may span up to one scale change.
        * If it does, the second portion of the ETH gain is scaled by 1e9.
        * If the gain spans no scale change, the second portion will be 0.
        */
        uint128 epochSnapshot = snapshots.epoch;
        uint128 scaleSnapshot = snapshots.scale;
        uint S_Snapshot = snapshots.S;
        uint P_Snapshot = snapshots.P;

        uint firstPortion = epochToScaleToSum[epochSnapshot][scaleSnapshot].sub(S_Snapshot);
        uint secondPortion = epochToScaleToSum[epochSnapshot][scaleSnapshot.add(1)].div(SCALE_FACTOR);

        uint ETHGain = initialDeposit.mul(firstPortion.add(secondPortion)).div(P_Snapshot).div(DECIMAL_PRECISION);

        return ETHGain;
    }
```

Description

***: The function `getDepositorETHGain` is used to calculate rewards based on the user's deposit:

```
uint ETHGain = initialDeposit.mul(firstPortion.add(secondPortion)).div(P_Snapshot).div(DECIMAL_PRE
CISION);
```

Here the **`initialDeposit`** is the amount of **`LUSD`** token.
The Magma protocal is forked from Liquity. In Liquity, the function **`getDepositorETHGain`** is used to calculate the ETH gain earned by the deposit since its last snapshots were taken. The decimal of ETH is 18. However, in Magma, the function **`getDepositorETHGain`** is changed to calculate the **`collToken`** gain:

```
function withdrawFromSP(uint _amount) external override {
        if (_amount !=0) {_requireNoUnderCollateralizedTroves();}
        uint initialDeposit = deposits[msg.sender].initialValue;
        _requireUserHasDeposit(initialDeposit);

        ICommunityIssuance communityIssuanceCached = communityIssuance;

        _triggerLQTYIssuance(communityIssuanceCached);

        uint depositorETHGain = getDepositorETHGain(msg.sender);

         ...
         ...

        _sendETHGainToDepositor(depositorETHGain);
    }

function _sendETHGainToDepositor(uint _amount) internal {
        if (_amount == 0) {return;}
        uint newETH = ETH.sub(_amount);
        ETH = newETH;
        emit StabilityPoolETHBalanceUpdated(newETH);
        emit EtherSent(msg.sender, _amount);

        if (isNativeToken(collToken)) {
            (bool success, ) = msg.sender.call{ value: _amount }("");
            require(success, "StabilityPool: sending ETH failed");
        } else {
            IERC20(collToken).transfer(msg.sender, _amount);
        }

    }
```

The issue here is that if the decimal of **`collToken`** is different from the decimal of **`LUSD`**, the calculated rewards will be wrong. For example, the **`collToken`** is **`USDC`** and the decimal of **`USDC`** is 6, then the rewards will be much more than expected.
***: As it is mentioned in the document:
This modification primarily focuses on adding support for ERC20 tokens to the existing running version, allowing users to collateralize ERC20 tokens to mint stablecoins(WEN).

But, the contract, `BorrowerOperations`, does not support ERC20 that has lower decimal, e.g. $WBTC(that has a decimal 8).
PoC:

- Admin add $WBTC(that has a decimal 8) as a collateral token;
- A user open a trove for 1000 $WEN with 0.1 $WBTC(1e7).
- In the `opernTrove` function, the arguments, the `_collToken` is $WBTC, the `_collAmount` is 1e7, the `_LUSDAmount` is 1000*1e18.
- Assume $WBTC price is 60000*1e18.
- Assume the `LUSDFee` is 5 $WEN. The `LUSD_GAS_COMPENSATION` is 1 $WEN.
- let's calculate the ICR per the `_computeCR` function: `1e7 * 60000*1e18 / (1006*1e18) = 596421471`, which is less than the default `MCR`, `1300000000000000000`. Finally fail to open the trove, but it should be success because 0.1 $WBTC worth $6000 and it should be success for the user to borrow 1000 $WEN.

Recommendation

***: Consider checking the decimal of `collToken` is 18:

```
function setCollToken(address _collToken) external onlyOwner {
        require(!isNativeToken(_collToken), "Invalid collToken");
        require(IERC20(_collToken).decimals() == 18,"wrong decimal");
        collToken = _collToken;
    }
```

***: Checking if the decimal equals to 18 when adding ERC20 token as collateral token.

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-4:Use safeTransferFrom instead of Transfer

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/BorrowerOperations.sol#L175
- code/contracts/BorrowerOperations.sol#L307
- code/contracts/BorrowerOperations.sol#L453
- code/contracts/BorrowerOperations.sol#L551

```
175: IERC20(_collToken).transferFrom(msg.sender, address(this), _collAmount);
```

```
307: IERC20(_collToken).transferFrom(msg.sender, address(this), _collChange);
```

```
453: lusdToken.transferFrom(msg.sender, address(this), _amount);
```

```
551: IERC20(_collToken).transfer(address(_activePool), _amount);
```

- code/contracts/CollTokenActivePool.sol#L101

```
101: IERC20(collToken).transfer(_account, _amount);
```

- code/contracts/CollTokenCollSurplusPool.sol#L109

```
109: IERC20(collToken).transfer(_account, claimableColl);
```

- code/contracts/CollTokenDefaultPool.sol#L87

```
87: IERC20(_collToken).transfer(activePoolAddress, _amount);
```

- code/contracts/CollTokenStabilityPool.sol#L879
- code/contracts/CollTokenStabilityPool.sol#L888

```
879: IERC20(collToken).transfer(msg.sender, _amount);
```

```
888: lusdToken.transfer(_depositor, LUSDWithdrawal);
```

Description

***: In contracts `BorrowerOperations` , `CollTokenActivePool` , `CollTokenCollSurplusPool` , `CollTokenDefaultPool` and `CollTokenStabilityPool` , the return value of the `transfer()` / `transferFrom()` call is not checked.

Recommendation

***: Since some ERC-20 tokens return no values and others return a bool value, they should be handled with care. We advise using the OpenZeppelin's `SafeERC20.sol` implementation to interact with the `transfer()` and `transferFrom()` functions of external ERC-20 tokens. The OpenZeppelin implementation checks for the existence of a return value and reverts if false is returned, making it compatible with all ERC-20 token implementations.

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-5:The `setAddresses` function does not check for duplicate `_collToken`

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/Dependencies/SysConfig.sol#L85-L121

```
85: function setAddresses(address _collToken,
86:                              address _troveManager,
87:                              address _sortedTroves,
88:                              address _surplusPool,
89:                              address _stabilityPool,
90:                              address _defaultPool,
91:                              address _activePool,
92:                              address _nativeTroveManager,
93:                              address _nativeTokenPriceFeed,
94:                              address _collTokenPriceFeed
95:     ) external onlyOwner {
96:         checkContract(_troveManager);
97:         checkContract(_sortedTroves);
98:         checkContract(_surplusPool);
99:         checkContract(_stabilityPool);
100:         checkContract(_defaultPool);
101:         checkContract(_activePool);
102:         checkContract(_nativeTroveManager);
103:         checkContract(_nativeTokenPriceFeed);
104:         checkContract(_collTokenPriceFeed);
```

```
105:
106:         tokenConfigData[_collToken].troveManager = _troveManager;
107:         tokenConfigData[_collToken].sortedTroves = _sortedTroves;
108:         tokenConfigData[_collToken].surplusPool = _surplusPool;
109:         tokenConfigData[_collToken].stabilityPool = _stabilityPool;
110:         tokenConfigData[_collToken].defaultPool = _defaultPool;
111:         tokenConfigData[_collToken].activePool = _activePool;
112:
113:         if (tokenConfigData[_collToken].mcr > 0 && !tokenConfigData[_collToken].enabled) {
114:             tokenConfigData[_collToken].enabled = true;
115:         }
116:         troveManagerPool[_troveManager] = true;
117:         nativeTokenTroveManager = ITroveManager(_nativeTroveManager);
118:         nativeTokenPriceFeed = IPriceFeed(_nativeTokenPriceFeed);
119:         collTokenPriceFeed = ICollTokenPriceFeed(_collTokenPriceFeed);
120:         collTokens.push(_collToken);
121:     }
```

Description

***: The `setAddresses` function will add `_collToken` to the `collTokens` array but does not check for duplicate tokens.

```
collTokens.push(_collToken);
```

The `collTokens` will be used in `getEntireSystemDebt` to calculate the entire system debt:

```
function getEntireSystemDebt() external view returns (uint entireSystemDebt) {
        uint totalDebt = LiquityBase(address(nativeTokenTroveManager)).getEntireSystemDebt();
        for (uint i = 0; i < collTokens.length; i++) {
            totalDebt += getEntireSystemDebt(collTokens[i]);
        }
        entireSystemDebt = totalDebt;
    }
```

If `collTokens` contains duplicate tokens, the result of the `getEntireSystemDebt` function will be wrong. The `entireSystemDebt` is used in `LiquityBase` to calculate `TCR`. If `TCR` is wrong, the main function like `adjustTrove` and `openTrove` in `BorrowerOperations` will be affected.

Recommendation

***: Consider following fix:

```
function setAddresses(address _collToken,
                          address _troveManager,
                          address _sortedTroves,
                          address _surplusPool,
                          address _stabilityPool,
                          address _defaultPool,
                          address _activePool,
                          address _nativeTroveManager,
                          address _nativeTokenPriceFeed,
                          address _collTokenPriceFeed
    ) external onlyOwner {
        checkContract(_troveManager);
        checkContract(_sortedTroves);
        checkContract(_surplusPool);
        checkContract(_stabilityPool);
        checkContract(_defaultPool);
        checkContract(_activePool);
        checkContract(_nativeTroveManager);
        checkContract(_nativeTokenPriceFeed);
        checkContract(_collTokenPriceFeed);

        tokenConfigData[_collToken].troveManager = _troveManager;
        tokenConfigData[_collToken].sortedTroves = _sortedTroves;
        tokenConfigData[_collToken].surplusPool = _surplusPool;
        tokenConfigData[_collToken].stabilityPool = _stabilityPool;
        tokenConfigData[_collToken].defaultPool = _defaultPool;
        tokenConfigData[_collToken].activePool = _activePool;

        if (tokenConfigData[_collToken].mcr > 0 && !tokenConfigData[_collToken].enabled) {
            tokenConfigData[_collToken].enabled = true;
        }
        troveManagerPool[_troveManager] = true;
        nativeTokenTroveManager = ITroveManager(_nativeTroveManager);
        nativeTokenPriceFeed = IPriceFeed(_nativeTokenPriceFeed);
        collTokenPriceFeed = ICollTokenPriceFeed(_collTokenPriceFeed);
        bool exits;
        for (uint i = 0; i < collTokens.length; i++) {
           if(collTokens[i] == _collToken){
              exits = true;
            }
         }
         require(!exits,"find duplicate token");
        collTokens.push(_collToken);
    }
```

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-6:The `onReceive` function does not check if the received token is the same with the `collToken`

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/CollTokenActivePool.sol#L174-L179

```
function onReceive(address _collToken, uint _amount) external override {
        _requireCallerIsBorrowerOperationsOrDefaultPool();

        ETH = ETH.add(_amount);
        emit ActivePoolCollTokenBalanceUpdated(_collToken, ETH);
    }
```

- code/contracts/CollTokenCollSurplusPool.sol#L134-L138

```
function onReceive(address _collToken, uint _amount) external override {
        _requireCallerIsActivePool();
        ETH = ETH.add(_amount);
        emit CollSurplusPoolCollTokenBalanceUpdated(_collToken, ETH);
    }
```

Description

***: In contract `CollTokenCollSurplusPool`, the `onReceive` function will add `_amount` to `ETH`:

```
function onReceive(address _collToken, uint _amount) external override {
        _requireCallerIsActivePool();
        ETH = ETH.add(_amount);
        emit CollSurplusPoolCollTokenBalanceUpdated(_collToken, ETH);
    }
```

The issue here is that it does not check the input param `_collToken` is the same with the `collToken` in `CollTokenCollSurplusPool`. If the input param `_collToken` is different from `collToken`, the amount `ETH` will be incorrectly increased.
The same issue exists in `CollTokenActivePool` contract.

Recommendation

***: Consider adding a check:

```
function onReceive(address _collToken, uint _amount) external override {
        _requireCallerIsActivePool();
        require(_collToken == collToken,"incorrect collToken");
        ETH = ETH.add(_amount);
        emit CollSurplusPoolCollTokenBalanceUpdated(_collToken, ETH);
    }
```

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-7:Missing Check the `msg.value` if the `_collToken` is not native token

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/BorrowerOperations.sol#L169-L237
- code/contracts/BorrowerOperations.sol#L302-L309

```
169: function openTrove(address _collToken, uint _collAmount, uint _maxFeePercentage, uint _LUSDAmount, address _upperHint, address _lowerHint) public payable {
170:         sysConfig.checkCollToken(_collToken);
171:
172:         if (isNativeToken(_collToken)) {
173:             _collAmount = msg.value;
174:         } else {
175:             IERC20(_collToken).transferFrom(msg.sender, address(this), _collAmount);
176:         }
177:         ITroveManager localTroveManager = _getTroveManager(_collToken);
178:         ContractsCache memory contractsCache = ContractsCache(localTroveManager, IActivePool(_getActivePool(_collToken)), lusdToken);
179:         LocalVariables_openTrove memory vars;
180:
181:         vars.price = sysConfig.fetchPrice(_collToken);
182:
183:         bool isRecoveryMode;
184:         if (isNativeToken(_collToken)) {
185:             isRecoveryMode = _checkRecoveryMode(vars.price);
186:         } else {
187:             isRecoveryMode = sysConfig.checkRecoveryMode(_collToken, vars.price);
188:         }
```

```
189:
190:         _requireValidMaxFeePercentage(_maxFeePercentage, isRecoveryMode);
191:         _requireTroveisNotActive(contractsCache.troveManager, msg.sender);
192:
193:         vars.LUSDFee;
194:         vars.netDebt = _LUSDAmount;
195:
196:         if (!isRecoveryMode) {
197:             vars.LUSDFee = _triggerBorrowingFee(contractsCache.troveManager, contractsCache.lusdToken, _LUSDAmount, _maxFeePercentage);
198:             vars.netDebt = vars.netDebt.add(vars.LUSDFee);
199:         }
200:         _requireAtLeastMinNetDebt(vars.netDebt);
201:
202:         // ICR is based on the composite debt, i.e. the requested LUSD amount + LUSD borrowing fee + LUSD gas comp.
203:         vars.compositeDebt = _getCompositeDebt(_collToken, vars.netDebt);
204:         assert(vars.compositeDebt > 0);
205:
206:         vars.ICR = LiquityMath._computeCR(_collAmount, vars.compositeDebt, vars.price);
207:         vars.NICR = LiquityMath._computeNominalCR(_collAmount, vars.compositeDebt);
208:
        if (isRecoveryMode) {
            _requireICRisAboveCCR(_collToken, vars.ICR);
        } else {
            _requireICRisAboveMCR(_collToken, vars.ICR);
            uint newTCR = _getNewTCRFromTroveChange(_collToken, _collAmount, true, vars.compositeDebt, true, vars.price);  // bools: coll increase, debt increase
            _requireNewTCRisAboveCCR(_collToken, newTCR);
        }

        // Set the trove struct's properties
        contractsCache.troveManager.setTroveStatus(msg.sender, 1);
        contractsCache.troveManager.increaseTroveColl(msg.sender, _collAmount);
        contractsCache.troveManager.increaseTroveDebt(msg.sender, vars.compositeDebt);

        contractsCache.troveManager.updateTroveRewardSnapshots(msg.sender);
        vars.stake = contractsCache.troveManager.updateStakeAndTotalStakes(msg.sender);

        ISortedTroves(_getSortedTroves(_collToken)).insert(msg.sender, vars.NICR, _upperHint, _lowerHint);
        vars.arrayIndex = contractsCache.troveManager.addTroveOwnerToArray(msg.sender);
        emit TroveCreated(msg.sender, address(localTroveManager), vars.arrayIndex);

```

```
        // Move the ether to the Active Pool, and mint the LUSDAmount to the borrower
        _activePoolAddColl(_collToken, contractsCache.activePool, _collAmount);
        _withdrawLUSD(contractsCache.activePool, msg.sender, _LUSDAmount, vars.netDebt);
        // Move the LUSD gas compensation to the Gas Pool
        _withdrawLUSD(contractsCache.activePool, gasPoolAddress, LUSD_GAS_COMPENSATION, LUSD_GAS_COMPENSATION);

        emit TroveUpdated(msg.sender, address(localTroveManager), vars.compositeDebt, _collAmount, vars.stake, BorrowerOperation.openTrove);
        emit LUSDBorrowingFeePaid(msg.sender, _collToken, vars.LUSDFee);
    }
```

```
function _adjustTrove(address _collToken, uint _collChange, address _borrower, uint _collWithdrawal, uint _LUSDChange, bool _isDebtIncrease, address _upperHint, address _lowerHint, uint _maxFeePercentage) internal {
        if (isNativeToken(_collToken)) {
            _collChange = msg.value;
        } else {
            if (_collChange > 0) {
                IERC20(_collToken).transferFrom(msg.sender, address(this), _collChange);
            }
        }
```

Description

***: In `openTrove` function, if the `_collToken` is a native token, the `_collAmount` will be replaced by the `msg.value`, otherwise, it will use the input param `_collAmount` directly:

```
function openTrove(address _collToken, uint _collAmount, uint _maxFeePercentage, uint _LUSDAmount,
address _upperHint, address _lowerHint) public payable {
        sysConfig.checkCollToken(_collToken);

        if (isNativeToken(_collToken)) {
            _collAmount = msg.value;
        } else {
            IERC20(_collToken).transferFrom(msg.sender, address(this), _collAmount);
        }
```

Another case ignored here is that if the `_collToken` is not a native token and `msg.value` is not 0, then this part of the ether will be locked in the current contract.
The same issue exists in `_adjustTrove` function.

Recommendation

***: Consider adding a check when the `_collToken` is not a native token:

```
        if (isNativeToken(_collToken)) {
            _collAmount = msg.value;
        } else {
            require(msg.value == 0,"invalid msg.value");
            IERC20(_collToken).transferFrom(msg.sender, address(this), _collAmount);
        }
```

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-8:Incompatible With the Deflationary or Fee-on-Transfer Tokens in Function

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/BorrowerOperations.sol#L169-L219
- code/contracts/BorrowerOperations.sol#L302-L309

```
169: function openTrove(address _collToken, uint _collAmount, uint _maxFeePercentage, uint _LUSDAmount, address _upperHint, address _lowerHint) public payable {
170:         sysConfig.checkCollToken(_collToken);
171:
172:         if (isNativeToken(_collToken)) {
173:             _collAmount = msg.value;
174:         } else {
175:             IERC20(_collToken).transferFrom(msg.sender, address(this), _collAmount);
176:         }
177:         ITroveManager localTroveManager = _getTroveManager(_collToken);
178:         ContractsCache memory contractsCache = ContractsCache(localTroveManager, IActivePool(_getActivePool(_collToken)), lusdToken);
179:         LocalVariables_openTrove memory vars;
180:
181:         vars.price = sysConfig.fetchPrice(_collToken);
182:
183:         bool isRecoveryMode;
184:         if (isNativeToken(_collToken)) {
185:             isRecoveryMode = _checkRecoveryMode(vars.price);
186:         } else {
187:             isRecoveryMode = sysConfig.checkRecoveryMode(_collToken, vars.price);
188:         }
```

```
189:
190:         _requireValidMaxFeePercentage(_maxFeePercentage, isRecoveryMode);
191:         _requireTroveisNotActive(contractsCache.troveManager, msg.sender);
192:
193:         vars.LUSDFee;
194:         vars.netDebt = _LUSDAmount;
195:
196:         if (!isRecoveryMode) {
197:             vars.LUSDFee = _triggerBorrowingFee(contractsCache.troveManager, contractsCache.lusdToken, _LUSDAmount, _maxFeePercentage);
198:             vars.netDebt = vars.netDebt.add(vars.LUSDFee);
199:         }
200:         _requireAtLeastMinNetDebt(vars.netDebt);
201:
202:         // ICR is based on the composite debt, i.e. the requested LUSD amount + LUSD borrowing fee + LUSD gas comp.
203:         vars.compositeDebt = _getCompositeDebt(_collToken, vars.netDebt);
204:         assert(vars.compositeDebt > 0);
205:
206:         vars.ICR = LiquityMath._computeCR(_collAmount, vars.compositeDebt, vars.price);
207:         vars.NICR = LiquityMath._computeNominalCR(_collAmount, vars.compositeDebt);
208:
        if (isRecoveryMode) {
            _requireICRisAboveCCR(_collToken, vars.ICR);
        } else {
            _requireICRisAboveMCR(_collToken, vars.ICR);
            uint newTCR = _getNewTCRFromTroveChange(_collToken, _collAmount, true, vars.compositeDebt, true, vars.price);  // bools: coll increase, debt increase
            _requireNewTCRisAboveCCR(_collToken, newTCR);
        }

        // Set the trove struct's properties
        contractsCache.troveManager.setTroveStatus(msg.sender, 1);
        contractsCache.troveManager.increaseTroveColl(msg.sender, _collAmount);
```

```
function _adjustTrove(address _collToken, uint _collChange, address _borrower, uint _collWithdrawal, uint _LUSDChange, bool _isDebtIncrease, address _upperHint, address _lowerHint, uint _maxFeePercentage) internal {
        if (isNativeToken(_collToken)) {
            _collChange = msg.value;
        } else {
            if (_collChange > 0) {
                IERC20(_collToken).transferFrom(msg.sender, address(this), _collChange);
            }
        }
```

Description

***: The function in the contract contains a critical flaw that
fails to correctly handle fee-on-transfer tokens. The contract assumes that the full amount of tokens will be transferred to the contract:

```
    function openTrove(address _collToken, uint _collAmount, uint _maxFeePercentage, uint _LUSDAmount, address _upperHint, address _lowerHint) public payable {
        sysConfig.checkCollToken(_collToken);

        if (isNativeToken(_collToken)) {
            _collAmount = msg.value;
        } else {
            IERC20(_collToken).transferFrom(msg.sender, address(this), _collAmount);//@audit deflationary token is not checked
        }
        ...
        vars.ICR = LiquityMath._computeCR(_collAmount, vars.compositeDebt, vars.price);
        vars.NICR = LiquityMath._computeNominalCR(_collAmount, vars.compositeDebt);
        ...
        contractsCache.troveManager.increaseTroveColl(msg.sender, _collAmount);
```

However, if the token deducts a fee on the transfer, the contract will not receive the full amount. As a result, when the function attempts to withdraw the same amount from the contract, the contract may not have enough tokens to perform all the withdrawals.

Recommendation

***:
Calculating the actual balance received by subtracting the balance of the token after the transfer to that of before the transfer, and applying the actual balance for the following business.

```
uint256 initialBalance = IERC20(_collToken).balanceOf(address(this));
IERC20(_collToken).transferFrom(msg.sender, address(this), _collAmount);
uint256 finalBalance = IERC20(_collToken).balanceOf(address(this));
uint256 actualReceived = finalBalance - initialBalance;
_collAmount = actualReceived;
...
```

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-9:If a trove is opened by a contract, it may not be closed any more

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/BorrowerOperations.sol#L404-L434

```
function closeTroveInternal(address collToken, address repayer, address troveOwner) internal {
        ITroveManager troveManagerCached = _getTroveManager(collToken);
        IActivePool activePoolCached = IActivePool(_getActivePool(collToken));
        ILUSDToken lusdTokenCached = lusdToken;

        _requireTroveisActive(troveManagerCached, troveOwner);
        uint price = sysConfig.fetchPrice(collToken);
        _requireNotInRecoveryMode(collToken, price);

        troveManagerCached.applyPendingRewards(troveOwner);

        uint coll = troveManagerCached.getTroveColl(troveOwner);
        uint debt = troveManagerCached.getTroveDebt(troveOwner);

        _requireSufficientLUSDBalance(lusdTokenCached, repayer, debt.sub(LUSD_GAS_COMPENSATION));

        uint newTCR = _getNewTCRFromTroveChange(collToken, coll, false, debt, false, price);
        _requireNewTCRisAboveCCR(collToken, newTCR);

        troveManagerCached.removeStake(troveOwner);
        troveManagerCached.closeTrove(troveOwner);

        emit TroveUpdated(troveOwner, address(troveManagerCached), 0, 0, 0, BorrowerOperation.closeTrove);

        // Burn the repaid LUSD from the user's balance and the gas compensation from the Gas Pool
        _repayLUSD(activePoolCached, repayer, debt.sub(LUSD_GAS_COMPENSATION));
        _repayLUSD(activePoolCached, gasPoolAddress, LUSD_GAS_COMPENSATION);

        // Send the collateral back to the user
        activePoolCached.sendETH(troveOwner, coll);
    }
```

- code/contracts/CollTokenActivePool.sol#L95-L105

```
function sendETH(address _account, uint _amount) external override {
        _requireCallerIsBOorTroveMorSP();
        ETH = ETH.sub(_amount);
        emit ActivePoolETHBalanceUpdated(ETH);
        emit EtherSent(_account, _amount);

         IERC20(collToken).transfer(_account, _amount);
         if (isContract(_account)) {
             ICollTokenReceiver(_account).onReceive(collToken, _amount);
         }
     }
```

Description

***: If a trove is opened by a contract, it may not be closed any more. This can happen when the `collToken` is an ERC20 token. Both `closeTroveOnBehalf` and `closeTrove` functions will call `closeTroveInternal` function, in the end of the `closeTroveInternal` function, it will call `sendETH` function:

```
// Send the collateral back to the user
activePoolCached.sendETH(troveOwner, coll);
```

If the `collToken` is an ERC20 token, the `activePoolCached` will be the `CollTokenActivePool` contract. In `CollTokenActivePool.sendETH()` function, it will call `onReceive` if the `_account` is an contract:

```
 function sendETH(address _account, uint _amount) external override {
        _requireCallerIsBOorTroveMorSP();
        ETH = ETH.sub(_amount);
        emit ActivePoolETHBalanceUpdated(ETH);
        emit EtherSent(_account, _amount);

        IERC20(collToken).transfer(_account, _amount);
        if (isContract(_account)) {
            ICollTokenReceiver(_account).onReceive(collToken, _amount);
        }
    }
```

If the `_account` implements neither the ICollTokenReceiver interface nor the fallback function, the `sendETH` function will revert and the `closeTroveInternal` function will revert too.
In summary, if a contract implements neither the ICollTokenReceiver interface nor the fallback function, the trove it creates will not be able to be closed.

Recommendation

***: When opening a trave, consider checking whether the depositor implements the `ICollTokenReceiver` interface if it is a contract.

Client Response

client response : Fixed. Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-10:Hardcoded TIMEOUT is not appropriate for all tokens

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/CollTokenPriceFeed.sol#L24

```
24: uint constant public TIMEOUT = 14400;  // 4 hours: 60 * 60 * 4
```

Description

***: The `CollTokenPriceFeed.sol` contract is designed to fetch and store the latest price of collateral tokens using Chainlink oracles. It implements a timeout mechanism to determine if the Chainlink data is stale or if the oracle is frozen. However, the timeout is hardcoded, which can lead to potential issues when dealing with different tokens that have varying heartbeat intervals.
The contract defines a constant `TIMEOUT` value, set to 4 hours. However, this does not account for the fact that different Chainlink price feeds have different heartbeat intervals. E.g. 1Inch/USD price feed has a heartbeat of 24 hours. If the price data is not updated within the last 4 hours, the contract will consider it stale and assume that the Chainlink oracle is frozen, even though the data might still be accurate.

```
Contract: CollTokenPriceFeed.sol

134:     function _chainlinkIsFrozen(ChainlinkResponse memory _response) internal view returns (bo
ol) {
135:         return block.timestamp.sub(_response.timestamp) > TIMEOUT;
136:     }
```

The same pattern is also used in other contracts (`PriceFeed.sol`) in `_chainlinkIsFrozen` functions.

Recommendation

***: We recommend adding a `heartbeat` mapping and relevant functions and querying the frozen state of the feed over it.

```
mapping (address => uint) public tokenHeartbeat;


function setTokenHeartbeat(address _collToken, uint _heartbeat) external onlyOwner {
    tokenHeartbeat[_collToken] = _heartbeat;
}

function _chainlinkIsFrozen(address _collToken, ChainlinkResponse memory _response) internal view
returns (bool) {
    uint timeout = tokenHeartbeat[_collToken];
    return block.timestamp.sub(_response.timestamp) > timeout;
}
```

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-11:Approve may revert if the `collToken` is USDT

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/CollTokenStabilityPool.sol#L452-491

```
452: function withdrawETHGainToTrove(address _upperHint, address _lowerHint) external override {
453:         uint initialDeposit = deposits[msg.sender].initialValue;
454:         _requireUserHasDeposit(initialDeposit);
455:         _requireUserHasTrove(msg.sender);
456:         _requireUserHasETHGain(msg.sender);
457:
458:         ICommunityIssuance communityIssuanceCached = communityIssuance;
459:
460:         _triggerLQTYIssuance(communityIssuanceCached);
461:
462:         uint depositorETHGain = getDepositorETHGain(msg.sender);
463:
464:         uint compoundedLUSDDeposit = getCompoundedLUSDDeposit(msg.sender);
465:         uint LUSDLoss = initialDeposit.sub(compoundedLUSDDeposit); // Needed only for event log
466:
467:         // First pay out any LQTY gains
468:         address frontEnd = deposits[msg.sender].frontEndTag;
469:         _payOutLQTYGains(communityIssuanceCached, msg.sender, frontEnd);
470:
471:         // Update front end stake
```

```
472:         uint compoundedFrontEndStake = getCompoundedFrontEndStake(frontEnd);
473:         uint newFrontEndStake = compoundedFrontEndStake;
474:         _updateFrontEndStakeAndSnapshots(frontEnd, newFrontEndStake);
475:         emit FrontEndStakeChanged(frontEnd, newFrontEndStake, msg.sender);
476:
477:         _updateDepositAndSnapshots(msg.sender, compoundedLUSDDeposit);
478:
479:         /* Emit events before transferring ETH gain to Trove.
480:          This lets the event log make more sense (i.e. so it appears that first the ETH gain is
withdrawn
481:         and then it is deposited into the Trove, not the other way around). */
482:         emit ETHGainWithdrawn(msg.sender, depositorETHGain, LUSDLoss);
483:         emit UserDepositChanged(msg.sender, compoundedLUSDDeposit);
484:
485:         ETH = ETH.sub(depositorETHGain);
486:         emit StabilityPoolETHBalanceUpdated(ETH);
487:         emit EtherSent(msg.sender, depositorETHGain);
488:
489:         IERC20(collToken).approve(address(borrowerOperations), depositorETHGain);
490:         borrowerOperations.moveCollTokenGainToTrove(collToken, depositorETHGain, msg.sender, _u
pperHint, _lowerHint);
491:     }
```

Description

***: The function `IERC20(collToken).approve()` is called in the function `withdrawETHGainToTrove()` to approve the allowance for the `address(borrowerOperations)`.

However, for the token like `USDT`, there are specialized checksums in the approve function.

```
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

So the call only success when `allowed` or `value` is 0. As a result, the call of function `IERC20(collToken).approve()` may fail when the previous allowed is not zero for `USDT`.

Recommendation

***: Recommend adding the call of `IERC20(collToken).approve(address(borrowerOperations), 0)`.

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-12:Unsafe casting

Category	Severity	Client Response	Contributor
Integer Overflow and Underflow	Informational	Fixed	***

Code Reference

- code/contracts/UniIOTXPrice.sol#L37-L39

```
37: uint exchangeRatio = iotxStaking.exchangeRatio();
38:             (, int currentIOTXPrice,,,) = iotxPriceOracle.latestRoundData();
39:             answer = int(exchangeRatio) * currentIOTXPrice / 1e18;
```

Description

***: In `UniIOTXPrice` contract, the value of `exchangeRatio` is an `uint`. This is cast to an `int` in the following calculation:

```
answer = int(exchangeRatio) * currentIOTXPrice / 1e18;
```

However, since `uint` can store higher values than `int`, it is possible that casting from `uint` to `int` may create an overflow.

Recommendation

***: Consider using OpenZeppelin's SafeCast contract

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-13:The error message is incorrect

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	***

Code Reference

- code/contracts/Dependencies/SysConfig.sol#L72

```
72: require(ccr > 0, "!mcr");
```

Description

***: In `updateConfig` function, the error message for the `ccr` (Collateral Coverage Ratio) validation is incorrect. The function currently uses the same error message for both `mcr` (Minimum Collateral Ratio) and `ccr` checks:

```
require(ccr > 0, "!mcr");
```

This results in an incorrect error message being displayed when the `ccr` validation fails.

Recommendation

***: Update the error message for the `ccr` validation to be specific to the `ccr` parameter. Also, ensure that `ccr` is greater than or equal to `mcr` to make the logic consistent in the protocol.

```
function updateConfig(address _collToken, uint mcr, uint ccr) external onlyOwner {
    require(mcr > 0, "!mcr");
-   require(ccr > 0, "!mcr");
+   require(ccr > 0, "!ccr");
+   require(ccr >= mcr, "CCR must be greater than or equal to MCR");
    tokenConfigData[_collToken].mcr = mcr;
    tokenConfigData[_collToken].ccr = ccr;
    if (tokenConfigData[_collToken].troveManager != address(0x0) && !tokenConfigData[_collToken].e
nabled) {
        tokenConfigData[_collToken].enabled = true;
    }
}
```

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

MAG-14:Redundant code

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	***

Code Reference

- code/contracts/CollTokenPriceFeed.sol#L137-L153

```
137: function _chainlinkPriceChangeAboveMax(ChainlinkResponse memory _currentResponse, ChainlinkResponse memory _prevResponse) internal pure returns (bool) {
138:         uint currentScaledPrice = _scaleChainlinkPriceByDigits(uint256(_currentResponse.answer), _currentResponse.decimals);
139:         uint prevScaledPrice = _scaleChainlinkPriceByDigits(uint256(_prevResponse.answer), _prevResponse.decimals);
140:
141:         uint minPrice = LiquityMath._min(currentScaledPrice, prevScaledPrice);
142:         uint maxPrice = LiquityMath._max(currentScaledPrice, prevScaledPrice);
143:
144:         /*
145:         * Use the larger price as the denominator:
146:         * - If price decreased, the percentage deviation is in relation to the the previous price.
147:         * - If price increased, the percentage deviation is in relation to the current price.
148:         */
149:         uint percentDeviation = maxPrice.sub(minPrice).mul(DECIMAL_PRECISION).div(maxPrice);
150:
151:         // Return true if price has more than doubled, or more than halved.
152:         return percentDeviation > MAX_PRICE_DEVIATION_FROM_PREVIOUS_ROUND;
153:     }
```

- code/contracts/CollTokenStabilityPool.sol#L326-L329
- code/contracts/CollTokenStabilityPool.sol#L868-L882

```
326: function setCollToken(address _collToken) external onlyOwner {
327:         require(!isNativeToken(_collToken), "Invalid collToken");
328:         collToken = _collToken;
329:     }
```

```
868: function _sendETHGainToDepositor(uint _amount) internal {
869:         if (_amount == 0) {return;}
870:         uint newETH = ETH.sub(_amount);
871:         ETH = newETH;
872:         emit StabilityPoolETHBalanceUpdated(newETH);
873:         emit EtherSent(msg.sender, _amount);
874:
875:         if (isNativeToken(collToken)) {
876:             (bool success, ) = msg.sender.call{ value: _amount }("");
877:             require(success, "StabilityPool: sending ETH failed");
878:         } else {
879:             IERC20(collToken).transfer(msg.sender, _amount);
880:         }
881:
882:     }
```

Description

***: The `_chainlinkPriceChangeAboveMax` function is an internal function and is not used in `CollTokenPriceFeed` contract.
***: In `_sendETHGainToDepositor` function, if the `collToken` is native token, it will send ether to the msg.sender:

```
if (isNativeToken(collToken)) {
            (bool success, ) = msg.sender.call{ value: _amount }("");
            require(success, "StabilityPool: sending ETH failed");
        }
```

However, in `setCollToken` function, the `collToken` is not allowed to be set to a native token:

```
function setCollToken(address _collToken) external onlyOwner {
        require(!isNativeToken(_collToken), "Invalid collToken");
        collToken = _collToken;
    }
```

As a result, the code for sending ether in `_sendETHGainToDepositor` function is never to be executed.

Recommendation

***: If this internal function is not intended to be used, consider removing it.
***: Consider following fix:

```
function _sendETHGainToDepositor(uint _amount) internal {
        if (_amount == 0) {return;}
        uint newETH = ETH.sub(_amount);
        ETH = newETH;
        emit StabilityPoolETHBalanceUpdated(newETH);
        emit EtherSent(msg.sender, _amount);

        IERC20(collToken).transfer(msg.sender, _amount);

    }
```

Client Response

client response : Fixed in commit 1095847b976534d90c093822b9fbf4d237ef8817.

Disclaimer