QuillAudits

## Overview

StackOS

StackOS is an open protocol that allows individuals to collectively offer a decentralized cloud where a user can deploy any full-stack application, decentralized app, blockchain privatenets, and mainnet nodes.

## Scope of Audit

The scope of this audit was to analyse **StackOS smart contract**'s codebase for quality, security, and correctness.

**StackOS Contracts**
**Commit**: 670ce91740088de4dfa3472a97dcc368d0fee3c4
**Fixed In**: 1356c1847a5f30dba14e6c4602f43ffd6ca7e87a

| | |
|---|---|
| Escrow<br><br>• BaseEscrow.sol<br>• EscrowStorage.sol<br>• IEscrow.sol<br>• StackEscrow.sol | The main payment contract in the system.<br><br>Individual users i.e. application deployers use this contract while deploying applications to make payments, recharge their account / renew their plan, and withdraw completely.<br><br>Structurally, escrow contracts have three parts:<br><br>**BaseEscrow**: Contains internal functions for creating a deposit, recharging accounts, and withdrawing funds and public functions for settling accounts<br><br>**StackEscrow**: Public functions for users and clusterOwners which call internal functions of BaseEscrow<br><br>**EscrowStorage**: Contains Storage Variables and Mappings |

| | |
|---|---|
| | Users can either recharge the account / upgrade the account or withdraw the deposit. For every action, first previous balance and calculations are settled, and then new changes are applied.<br><br>It calculates how many funds are utilised so far using totalDripRate for the deposit. Then subtracts this utilised amount from totalDeposit.<br><br>Utilised funds are transferred to the cluster Owner. |
| uniswap | Uniswap Interfaces for Factory, Router and Pair |
| Cluster-metadata<br>● DnsClusterMetadataStore.sol<br>● IDnsClusterMetadataStore.sol | This contract stores all the metadata of a cluster like owner, IP Address, upvotes, downvotes, isActive, qualityFactor, etc. A new cluster entry can be created only by the staking contract. Each cluster has functions to upvote and downvote the cluster. A cluster can also be marked defaulter by the owner. |
| Resource-feed<br>● ResourceFeed.sol<br>● IResourceFeed.sol | It keeps track of all the resources of a cluster. Resources can be added, removed and updated by the cluster owners. Each resource has a drip rate and a voting weight associated with it, which can be updated by the cluster owner. |
| Oracle<br>● Oracle.sol<br>● IPriceOracle.sol | Oracle's role is to recompute the average price for the entire period once every period for a longer time for token conversion via Uniswap |
| Staking<br>● Staking.sol | Staking role is to set staking amount to the depositor, stake token to the cluster, withdrawal of staker amount along with calculation of staked share percentage of the invoker. |

## Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC20 transfer() does not return boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

## Techniques and Methods

Throughout the audit of **StackOS smart contracts,** care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

### Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the Smart contract is structured in a way that will not result in future problems.

### Static Analysis

Static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

### Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

### Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

## Tools and Platforms used for Audit

Mythril, Slither, SmartCheck, Surya, Solhint.

## Issue Categories

Every issue in this report has been assigned with a severity level. There are four levels of severity, and each of them has been explained below.

### High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues to be fixed before moving to a live environment.

### Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

### Low Severity Issues

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

### Informational

These are four issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

## Number of issues per severity

| TYPE | HIGH | MEDIUM | LOW | INFORMATIONAL |
|---|---|---|---|---|
| Open | 0 | 0 | 0 | 0 |
| Acknowledged | 0 | 0 | 3 | 15 |
| Closed | 6 | 4 | 1 | 4 |

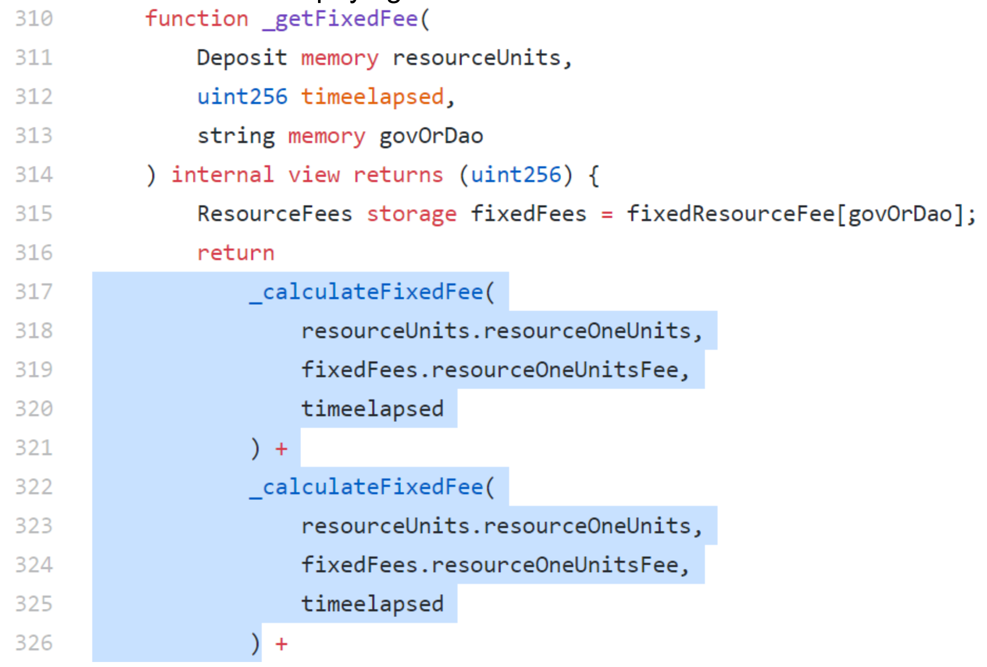
## Issues Found

## High Severity Issues

### BaseEscrow & StackEscrow

- **Calculating FixedFee for Same Resource Twice**

  **[#L310-362] _getFixedFee function** calculates fixedFees for resourceOneUnits twice, as a result a user will be paying more fees than intended.



```
310     function _getFixedFee(
311         Deposit memory resourceUnits,
312         uint256 timeelapsed,
313         string memory govOrDao
314     ) internal view returns (uint256) {
315         ResourceFees storage fixedFees = fixedResourceFee[govOrDao];
316         return
317             _calculateFixedFee(
318                 resourceUnits.resourceOneUnits,
319                 fixedFees.resourceOneUnitsFee,
320                 timeelapsed
321             ) +
322             _calculateFixedFee(
323                 resourceUnits.resourceOneUnits,
324                 fixedFees.resourceOneUnitsFee,
325                 timeelapsed
326             ) +
```

**Recommendation:**

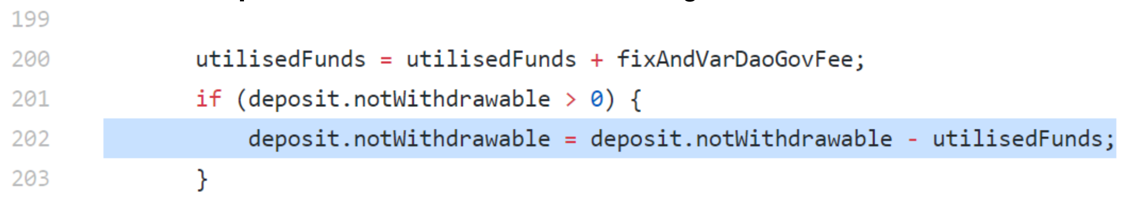
Remove the duplicate statements and Fix the fees Calculation

**Status: Fixed**
The fixed fees have been removed entirely.

## Underflow leading to Unoperational Withdrawal for Depositor

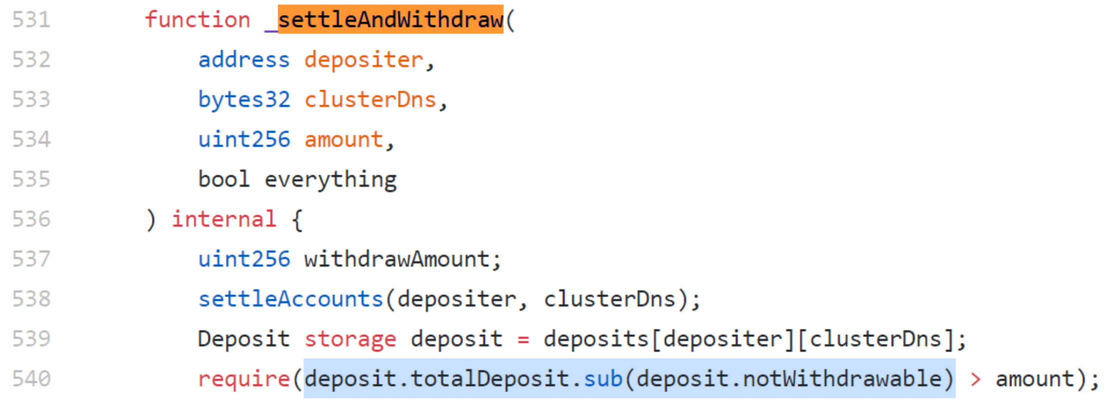
**[#L159-218] settleAccounts function:** The function checks, if there is any notWithdrawable balance in the depositor's account, and subtracts utilisedFunds from it.

As **utilisedFunds** can be more than **notWithdrawable**, it can lead to integer underflow, which will make **deposit.notWithdrawable** in the range of **$2^{256}$**.



```
199
200            utilisedFunds = utilisedFunds + fixAndVarDaoGovFee;
201            if (deposit.notWithdrawable > 0) {
202                deposit.notWithdrawable = deposit.notWithdrawable - utilisedFunds;
203            }
```

So, if a user tries to withdraw its **withdrawable** balance with **[#L531-572] _settleAndWithdraw function**, the SafeMath subtraction of **notWithdrawable** balance from **totalDeposit** will fail, resulting in failure of withdrawal of withdrawable balance.



```
531        function _settleAndWithdraw(
532            address depositer,
533            bytes32 clusterDns,
534            uint256 amount,
535            bool everything
536        ) internal {
537            uint256 withdrawAmount;
538            settleAccounts(depositer, clusterDns);
539            Deposit storage deposit = deposits[depositer][clusterDns];
540            require(deposit.totalDeposit.sub(deposit.notWithdrawable) > amount);
```

**Possible Exploit Scenarios:**

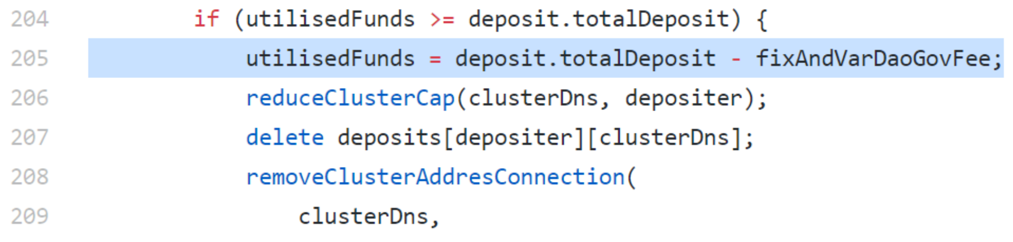
1. Owner issues a grant to an account, thereby increasing the notWithdrawable balance for that account, and if the depositor has funds or later decides to recharge its account. With this the user can utilize more funds than notWithdrawable balance, thus leading to underflow

2. ClusterOwner can rebate an account by sending some notWithdrawable balance, to an account. This can be intentional as well, taking advantage of underflow

**Status:** Fixed
The code has been updated with the logic to deduct **utlisedFunds** from **notWithdrawable** balance, only if the **notWithdrawable** balance is more than the **utlisedFunds**, and if it is less than **utlisedFunds**, then it means the **notWithdrawable** has been completely utilized and hence updated to 0.

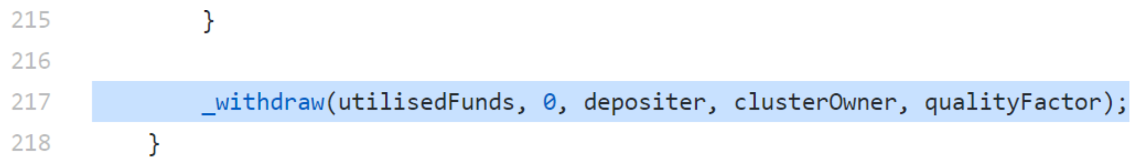
- **Incorrect Fees Collection can lead to Underflow, leading to Unoperational Withdrawal for ClusterOwner, Unoperational Settling of Account or possible drain of all available contract balance.**

  **[#L159-218] settleAccounts function:** As function deducts fixed and variable fees for dao and gov addresses. Incorrect fixed and variable fees settings may become more than the utilisedFunds itself, thereby subtraction of **fixAndVarDaoGovFee** from **totalDeposit** will cause an underflow.



```
204            if (utilisedFunds >= deposit.totalDeposit) {
205                utilisedFunds = deposit.totalDeposit - fixAndVarDaoGovFee;
206                reduceClusterCap(clusterDns, depositer);
207                delete deposits[depositer][clusterDns];
208                removeClusterAddresConnection(
209                    clusterDns,
```

And as the function sends the utilisedFunds to ClusterOwner



```
215            }
216
217            _withdraw(utilisedFunds, 0, depositer, clusterOwner, qualityFactor);
218        }
```

Theoretically, it may drain out/ transfer all the contract's balance to ClusterOwner, or logically, the operation will fail as the contract will never be having this much balance, as a result **settleAccount** will fail.

______**Possible Exploit Scenarios:**

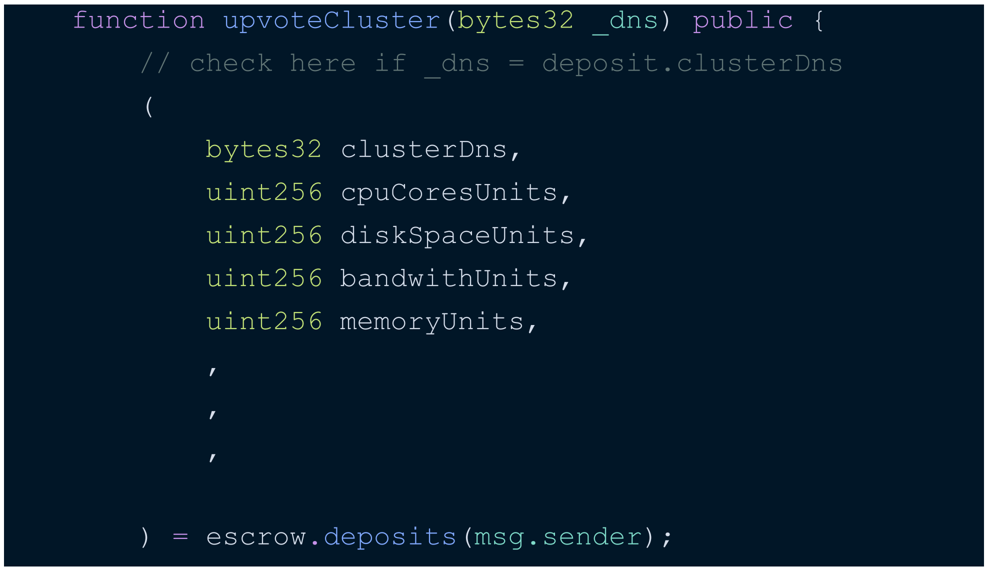
1. The first scenario has already been described above.

2. **[In Theory]** An attacker can set up a cluster from **staking** contract, deposit some funds as a user, and can call **settleAccount** in order to drain the contract balance.

**Status:** Fixed
The fixed fees have been removed entirely

## DnsClusterMetadataStore

- ### No deposits() function in Escrow contracts

| Line | Code/Function |
|---|---|
| 117-128 |  |

```
    function upvoteCluster(bytes32 _dns) public {
        // check here if _dns = deposit.clusterDns
        (
            bytes32 clusterDns,
            uint256 cpuCoresUnits,
            uint256 diskSpaceUnits,
            uint256 bandwithUnits,
            uint256 memoryUnits,
            ,
            ,
            ,

        ) = escrow.deposits(msg.sender);
```

**Description**

There is no function with the name 'deposits' in the contract files related to Escrow. This function is also defined in the IEscrow interface. The upvoteCluster() function will always revert because there is no 'deposits' function to be called.

**Recommendation**

Add the missing function in the Escrow contract and update it accordingly in the interface.

**Status: Fixed**

A function getDeposits() was added in the Escrow contracts.

- **DNS Cluster entry can be replaced by any user**

| Line | Code/Function |
|---|---|
| 74-96 | <pre>function addDnsToClusterEntry(<br>        bytes32 _dns,<br>        address _clusterOwner,<br>        string memory _ipAddress,<br>        string memory _whitelistedIps,<br>        string memory _clusterType,<br>        bool _isPrivate<br>    ) public onlyStakingContract {<br>        ClusterMetadata memory metadata = ClusterMetadata(<br>            _clusterOwner,<br>            _ipAddress,<br>            _whitelistedIps,<br>            0,<br>            0,<br>            false,<br>            100,<br>            true,<br>            _clusterType,<br>            _isPrivate<br>        );<br><br>        dnsToClusterMetadata[_dns] = metadata;<br>    }</pre> |

**Description**

The addDnsToClusterEntry() function can only be called by the staking contract. In the staking contract, any user can stake some STACK tokens and create a new DNS entry in the cluster metadata contract, by calling the deposit() function. There is no check if the ‘dns’ entry is already in the cluster metadata. It will always overwrite the previous entries.

This can be used by any malicious user to gain the cluster ownership and manipulate resource values according to their benefit.

**Recommendation**

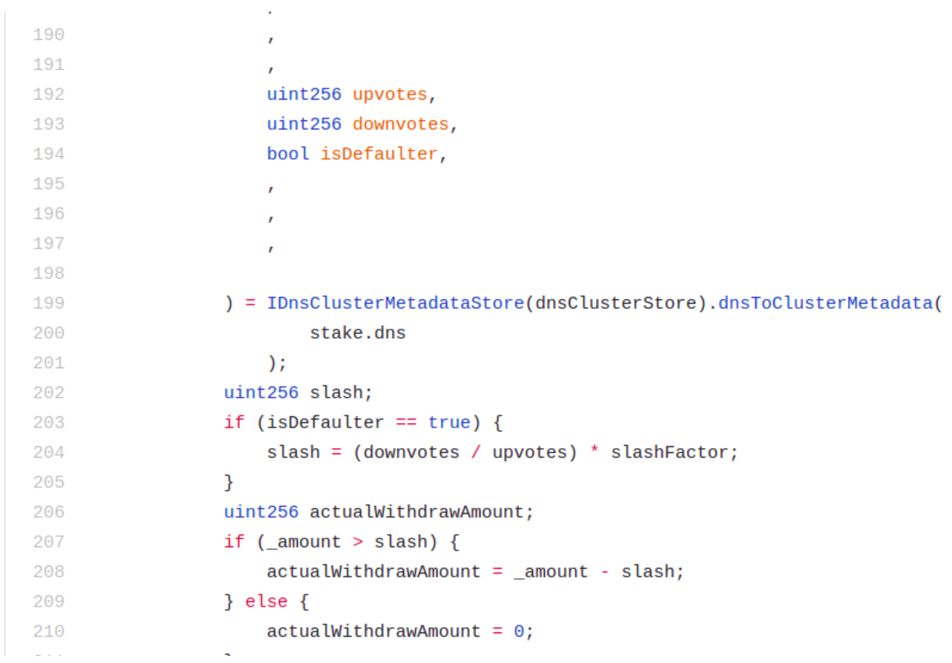
Check for previous entries of the ‘dns’ before adding a new one.

**Status: Fixed**
A require check was added before creating a new dns entry.

## Oracle (Oracle.sol, IPriceOracle.sol, Staking.sol)

- **Edge case for the divide isn't handled properly for the withdraw function.**

While calculating the slash value, there is a chance that upvotes for the cluster is 0 and dividing downvotes with 0 will result in an error.



```
            ,
            ,
            uint256 upvotes,
            uint256 downvotes,
            bool isDefaulter,
            ,
            ,
            ,

        ) = IDnsClusterMetadataStore(dnsClusterStore).dnsToClusterMetadata(
                stake.dns
            );
        uint256 slash;
        if (isDefaulter == true) {
            slash = (downvotes / upvotes) * slashFactor;
        }
        uint256 actualWithdrawAmount;
        if (_amount > slash) {
            actualWithdrawAmount = _amount - slash;
        } else {
            actualWithdrawAmount = 0;
```

Which means if isDefaulter == true and upvotes == 0 then the staker can never withdraw.

**Recommendation**

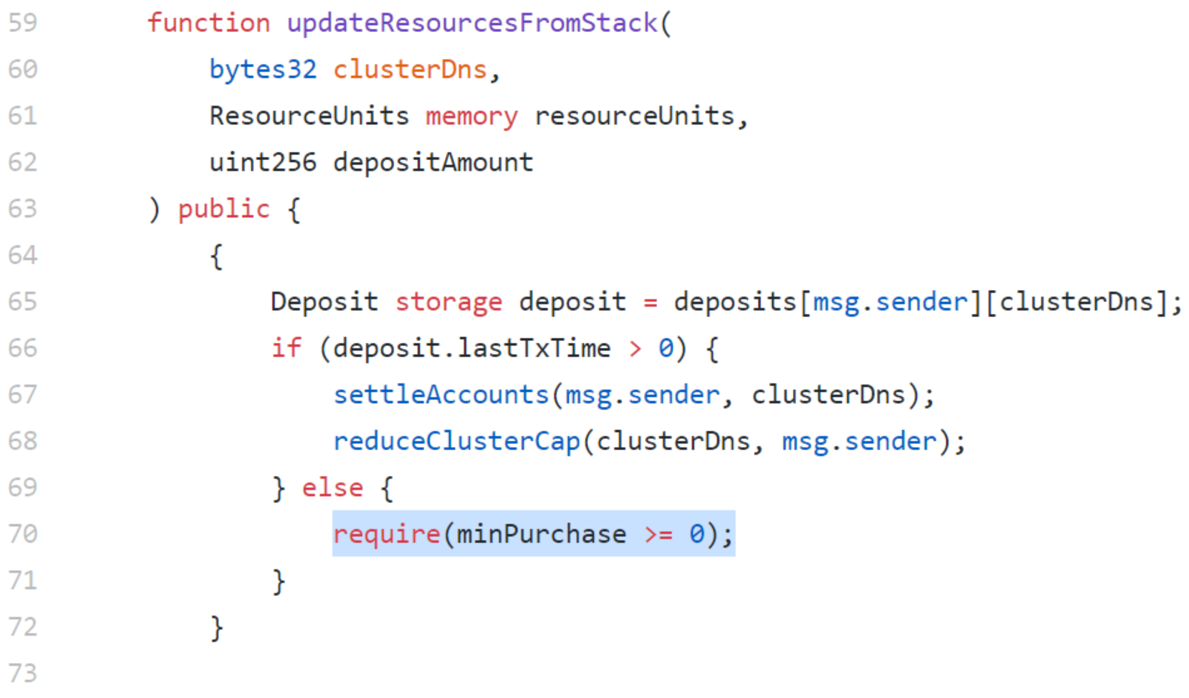
We need to define the scenario for this edge case.

**Status: Fixed**

## Medium Severity Issues

### BaseEscrow & StackEscrow

- **Code with no Logic**
  **[#L59-82] updateResourcesFromStack function:** It checks for an existing deposit entry by checking the lastTxTime, and if there is no entry, the else condition checks for **minPurchase >= 0** which will be true all the time, and thus have no logic.



```
59        function updateResourcesFromStack(
60            bytes32 clusterDns,
61            ResourceUnits memory resourceUnits,
62            uint256 depositAmount
63        ) public {
64            {
65                Deposit storage deposit = deposits[msg.sender][clusterDns];
66                if (deposit.lastTxTime > 0) {
67                    settleAccounts(msg.sender, clusterDns);
68                    reduceClusterCap(clusterDns, msg.sender);
69                } else {
70                    require(minPurchase >= 0);
71                }
72            }
73
```

**Recommendation:** A possible implementation can be, to check depositAmount should be greater than or equal to minPurchase.

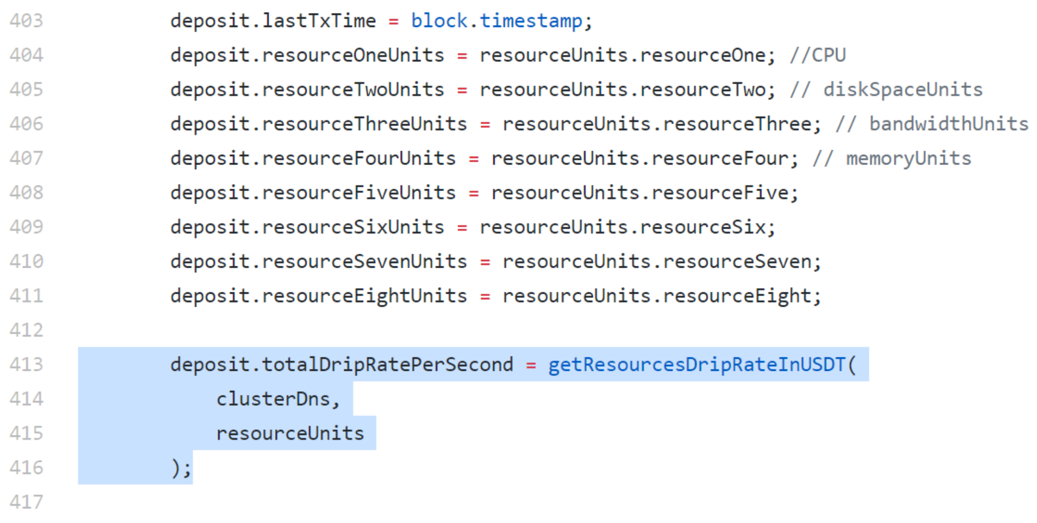
**Status:** **Fixed**
The code has now been updated with implementing the recommended logic stated above.

## ● 0 totalDripRatePerSecond can lead to Unoperational settleAccounts

**[#L387-424] _createDepositInternal function:** The function calculates **totalDripRatePerSecond** for a deposit as the accumulated drip rate for all the resources that a depositor has taken.

Drip Rate for one resource is calculated as **dripRatePerUnit * resourceUnits**

Where resourceUnits defines the number of units for a particular resource.



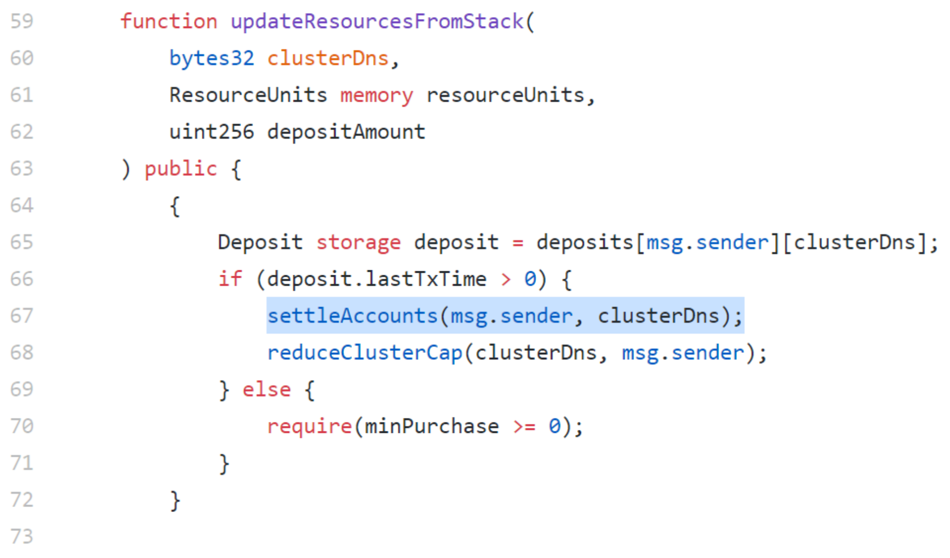
```
        deposit.lastTxTime = block.timestamp;
        deposit.resourceOneUnits = resourceUnits.resourceOne; //CPU
        deposit.resourceTwoUnits = resourceUnits.resourceTwo; // diskSpaceUnits
        deposit.resourceThreeUnits = resourceUnits.resourceThree; // bandwidthUnits
        deposit.resourceFourUnits = resourceUnits.resourceFour; // memoryUnits
        deposit.resourceFiveUnits = resourceUnits.resourceFive;
        deposit.resourceSixUnits = resourceUnits.resourceSix;
        deposit.resourceSevenUnits = resourceUnits.resourceSeven;
        deposit.resourceEightUnits = resourceUnits.resourceEight;

        deposit.totalDripRatePerSecond = getResourcesDripRateInUSDT(
            clusterDns,
            resourceUnits
        );

```

So, in cases, where drip rates for resources have not yet been set, or the user accidentally has taken 0 **resourceUnits**, the **totalDripRatePerSecond** calculated will be 0.

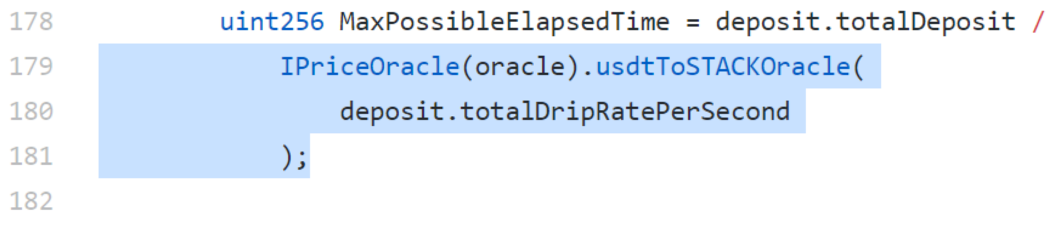
If a user tries to update the resources thereafter by calling **[#L59-82] updateResourcesFromStack function** in StackEscrow, it will never work for the depositor, as the function firstly tries to call **settleAccounts** for the depositor.



```
    function updateResourcesFromStack(
        bytes32 clusterDns,
        ResourceUnits memory resourceUnits,
        uint256 depositAmount
    ) public {
        {
            Deposit storage deposit = deposits[msg.sender][clusterDns];
            if (deposit.lastTxTime > 0) {
                settleAccounts(msg.sender, clusterDns);
                reduceClusterCap(clusterDns, msg.sender);
            } else {
                require(minPurchase >= 0);
            }
        }

```

And settleAccounts is now unoperational for the depositor due to **divide by 0 issue ( totalDripRatePerSecond = 0 )**



```
            uint256 MaxPossibleElapsedTime = deposit.totalDeposit /
                IPriceOracle(oracle).usdtToSTACKOracle(
                    deposit.totalDripRatePerSecond
                );

```

For the same reason [#L580-586] **settleMultipleAccounts** may never work

**Status: Fixed**

The divide by 0 issue has now been fixed.

## DnsClusterMetadataStore

- **The vote ratio can be manipulated**

| Line | Code/Function |
|---|---|
| 146-147 | `clusterUpvotes[_dns][msg.sender] = clusterUpvotes[_dns][msg.sender] + 1;` `dnsToClusterMetadata[_dns].upvotes += 1;` |

**Description**

The ratio of downvotes to upvotes is calculated and used in the Staking Contract as a ‘slash’ value, when the cluster is declared a defaulter. Now this ratio can be manipulated by any user by upvoting and then downvoting a cluster, several times.

**Recommendation**

Change the logic implementation for upvote and downvote of the cluster.

**Status: Fixed**
Voting logic was changed so that no single user can manipulate the votes.

- **getTotalVotes() reverts if voting weights are not set for the resource**

| Line | Code/Function |
|---|---|
| 638 | <pre>    function _calculateVotesPerResource(<br>        bytes32 clusterDns,<br>        string calldata name,<br>        uint256 resourceUnits<br>    ) internal view returns (uint256) {<br>        return<br>            (resourceUnits * 1e18) /<br>         resourceFeed.getResourceVotingWeight(clusterDns,<br>name);<br>    }</pre> |

**Description**

Resource units are divided by the voting weights of that specific resource. If the voting weights are not set for that resource then it is 0 by default,which means resourceUnits will be divided by 0. This transaction will revert. The upvoteCluster() function uses this function to calculate the votingCapacity, which will also revert.

**Remediation**

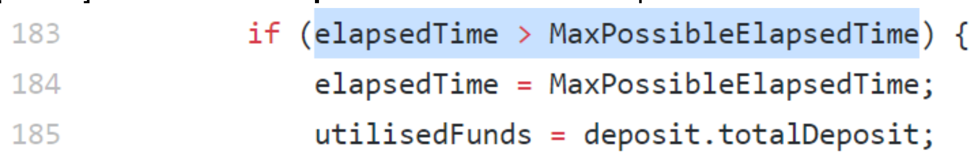
Keep a require check for the case when voting weights are zero.

**Status: Fixed**
A require check for the case ‘when voting weights are zero’ was added.

## Low Severity Issues

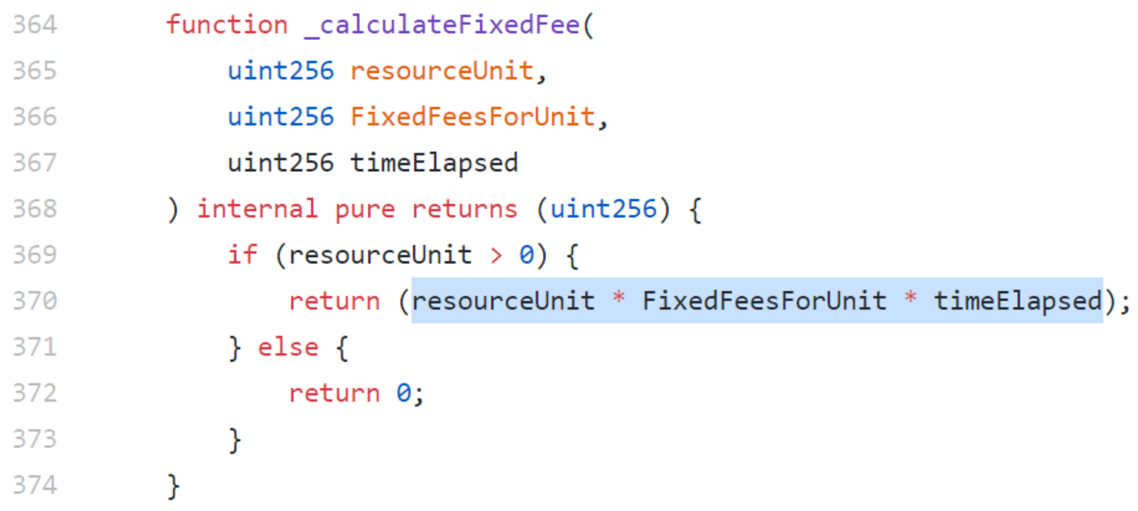
### BaseEscrow & StackEscrow

- [#L183] **block.timestamp** has been used for comparison



```
183             if (elapsedTime > MaxPossibleElapsedTime) {
184                 elapsedTime = MaxPossibleElapsedTime;
185                 utilisedFunds = deposit.totalDeposit;
```

And for calculating FixedFees for dao and gov



```
364         function _calculateFixedFee(
365             uint256 resourceUnit,
366             uint256 FixedFeesForUnit,
367             uint256 timeElapsed
368         ) internal pure returns (uint256) {
369             if (resourceUnit > 0) {
370                 return (resourceUnit * FixedFeesForUnit * timeElapsed);
371             } else {
372                 return 0;
373             }
374         }
```

**Recommendation**

Avoid using **block.timestamp**, as it can be manipulated by miners.

**Status:** **Acknowledged by the Auditee.**

- **Missing Zero Address Validation**

  **[#L50-74] constructor()** missing zero address checks for **_stackToken, _resourceFeed, _staking, _dnsStore, _factory, _router, _dao, _gov, _weth, _usdt, _oracle** addresses

  **[#L122-128] setFeeAddress()** missing zero address checks for **_daoAddress** and **_govAddress** addresses

______**Status:** **Acknowledged by the Auditee.**
It will be fixed in a future release

## DnsClusterMetadataStore

- **Hardcoded strings for resource names**

| Line | Code |
|---|---|
| 198-207 | `votes = _calculateVotesPerResource(clusterDns, "cpu", cpuCoresUnits);` `votes = votes + _calculateVotesPerResource(clusterDns, "memory", memoryUnits);` `votes = votes + _calculateVotesPerResource(clusterDns, "bandwidth", bandwidthUnits);` `votes = votes + _calculateVotesPerResource(clusterDns, "disk", diskSpaceUnits);` |

**Description**

Cluster owners can add resources with any name in the ResourceFeed contract. In the DnsClusterMetadataStore contract, getTotalVotes() function expects resources with some specific names to be present. If the clusterOwner never adds these resources then, upvoteCluster() will always revert.

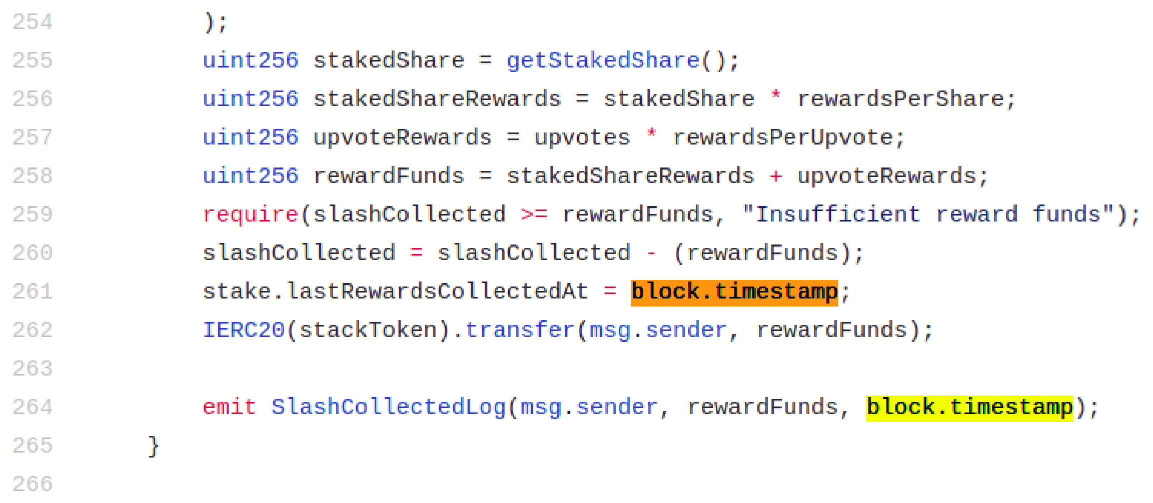
**Remediation**

Do not use predefined names for resources, in votes calculation. Or make sure that cluster owners add resources with the same name in the ResourceFeed Contract.

**Status: Fixed**
Resource names are now retrieved from the Escrow contract.

## Oracle (Oracle.sol, IPriceOracle.sol, Staking.sol)

- **Potential use of "block.timestamp" as a source of randomness (staking.sol)**



```
        );
        uint256 stakedShare = getStakedShare();
        uint256 stakedShareRewards = stakedShare * rewardsPerShare;
        uint256 upvoteRewards = upvotes * rewardsPerUpvote;
        uint256 rewardFunds = stakedShareRewards + upvoteRewards;
        require(slashCollected >= rewardFunds, "Insufficient reward funds");
        slashCollected = slashCollected - (rewardFunds);
        stake.lastRewardsCollectedAt = block.timestamp;
        IERC20(stackToken).transfer(msg.sender, rewardFunds);

        emit SlashCollectedLog(msg.sender, rewardFunds, block.timestamp);
    }

```

Contracts often need access to time values to perform certain types of functionality. Values such as block.timestamp, and block.number can give you a sense of the current time or a time delta, however, they are not safe to use for most purposes.

In the case of block.timestamp, developers often attempt to use it to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. Moreover, malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. However, miners can't set a timestamp smaller than the previous one (otherwise the block will be rejected), nor can they set the timestamp too far ahead in the future. Taking all of the above into consideration, developers can't rely on the preciseness of the provided timestamp.

**Remediation**

Developers should write smart contracts with the notion that block values are not precise, and the use of them can lead to unexpected effects. Alternatively, they may make use of oracles.

**References**

- Safety: Timestamp dependence
- Ethereum Smart Contract Best Practices - Timestamp Dependence
- How do Ethereum mining nodes maintain a time consistent with the network?
- Solidity: Timestamp dependency, is it possible to do safely?

**Status: Acknowledged by the Auditee.**

## Informational

- Public functions that are never called by the contract should be declared external to save gas.

  **Status:** **Acknowledged by the Auditee.**

- An old compiler has been used.

  **Recommendation**

  Use the latest solidity compilers so as to avoid bugs introduced in older versions

  **Status:** **Acknowledged by the Auditee.**

- Multiple Pragma Directives have been used.

  **Recommendation:** Use one solidity compiler

  **Status:** **Acknowledged by the Auditee.**
  It will be fixed in a future release.

## BaseEscrow & StackEscrow

- **No Penalty Deduction due to 100% quality factor**

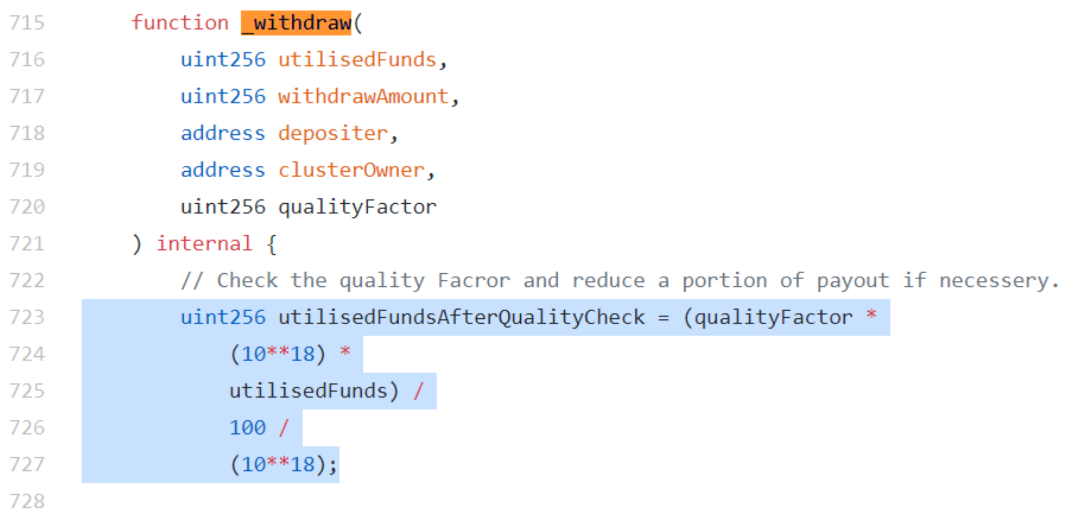
  The clusters are always set up with **qualityFactor** as 100.

```
    function addDnsToClusterEntry(
        bytes32 _dns,
        address _clusterOwner,
        string memory _ipAddress,
        string memory _whitelistedIps,
        string memory _clusterType,
        bool _isPrivate
    ) public onlyStakingContract {
        ClusterMetadata memory metadata = ClusterMetadata(
            _clusterOwner,
            _ipAddress,
            _whitelistedIps,
            0,
            0,
            false,
            100,
            true,
            _clusterType,
            _isPrivate
        );

        dnsToClusterMetadata[_dns] = metadata;
    }
```

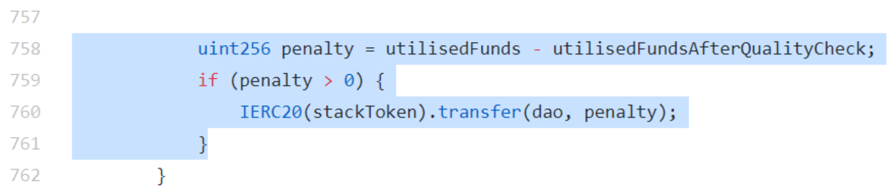
And there are no features available to modify the qualityFactor later on.

The **utilisedFundsAfterQualityCheck** will always be the same as **utilisedFunds**



```
715    function _withdraw(
716        uint256 utilisedFunds,
717        uint256 withdrawAmount,
718        address depositer,
719        address clusterOwner,
720        uint256 qualityFactor
721    ) internal {
722        // Check the quality Facror and reduce a portion of payout if necessery.
723        uint256 utilisedFundsAfterQualityCheck = (qualityFactor *
724            (10**18) *
725            utilisedFunds) /
726            100 /
727            (10**18);
728
```

As a result, the penalty in [#L715-767] **_withdraw** function will always be 0, hence will never be deducted.



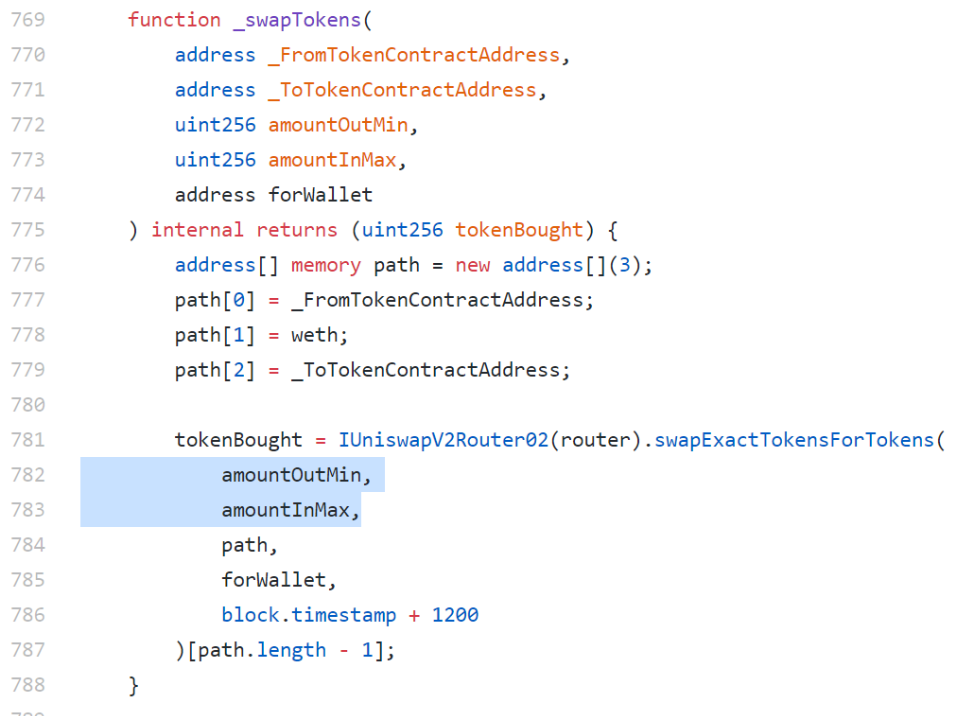
```
757
758            uint256 penalty = utilisedFunds - utilisedFundsAfterQualityCheck;
759            if (penalty > 0) {
760                IERC20(stackToken).transfer(dao, penalty);
761            }
762        }
```

**Status: Acknowledged by the Auditee.**

No penalty deduction will happen in the current release of the Contracts, as the clusters will all be from the StackOS itself.

- **Incorrect Parameters**

**[#L769-788] _swapTokens function** takes input value of the token in parameter **amountOutMin** and minimum output amount value in **amountInMax**. Although the values passed to UniswapRouter for swapping are correct, the parameter names give different meanings and create confusion in code readability.



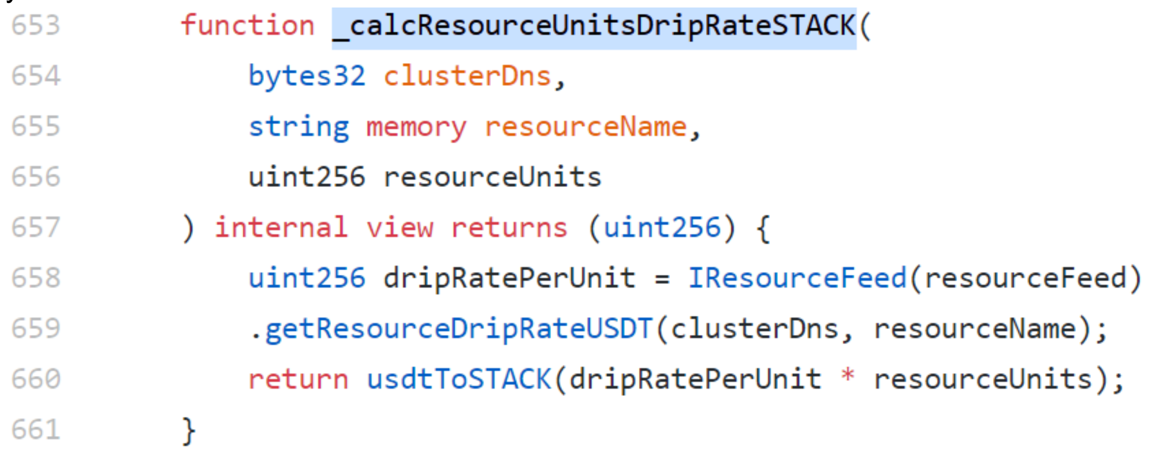
```
769    function _swapTokens(
770        address _FromTokenContractAddress,
771        address _ToTokenContractAddress,
772        uint256 amountOutMin,
773        uint256 amountInMax,
774        address forWallet
775    ) internal returns (uint256 tokenBought) {
776        address[] memory path = new address[](3);
777        path[0] = _FromTokenContractAddress;
778        path[1] = weth;
779        path[2] = _ToTokenContractAddress;
780
781        tokenBought = IUniswapV2Router02(router).swapExactTokensForTokens(
782            amountOutMin,
783            amountInMax,
784            path,
785            forWallet,
786            block.timestamp + 1200
787        )[path.length - 1];
788    }
```

**Recommendation:** Rename parameters with their logical meaning.

**Status: Fixed**

- **Dead Code**

**[#L653-661] _calcResourceUnitsDripRateSTACK** internal function has not been used anywhere in the contract



```
653        function _calcResourceUnitsDripRateSTACK(
654            bytes32 clusterDns,
655            string memory resourceName,
656            uint256 resourceUnits
657        ) internal view returns (uint256) {
658            uint256 dripRatePerUnit = IResourceFeed(resourceFeed)
659            .getResourceDripRateUSDT(clusterDns, resourceName);
660            return usdtToSTACK(dripRatePerUnit * resourceUnits);
661        }
```

**Status:** **Acknowledged by the Auditee.**

- **Comparison with a boolean constant.**

| | |
|---|---|
| #L397 | active == true |
| #L419 | grant == false |
| #L420 & #L525 | withdrawable == false |
| #L512 & #L528 | OverLimit == false |
| #L553 | everything == false |

**Recommendation:**

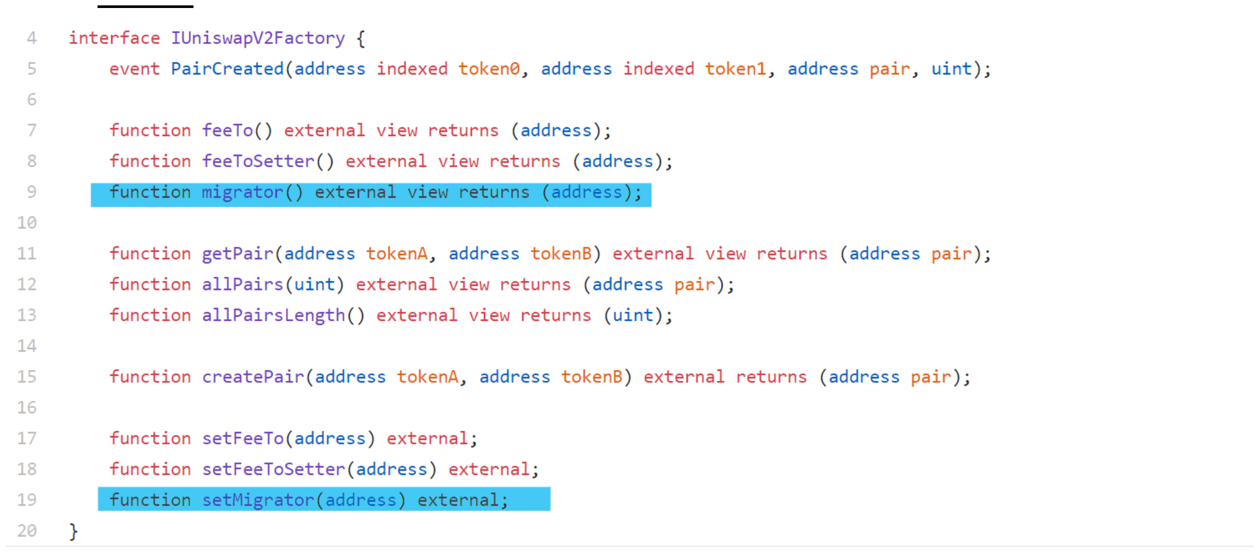
Boolean constants can be used directly and do not need to be compared to true or false.

**Status:** **Acknowledged by the Auditee.**

The devs will stick to the existing coding style.

## IUniswapV2Factory

- The Interface has some unknown functions as compared to the official Uniswap V2 Factory interface for which the details are not provided in the contract specification document.



```
interface IUniswapV2Factory {
    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    function feeTo() external view returns (address);
    function feeToSetter() external view returns (address);
    function migrator() external view returns (address);

    function getPair(address tokenA, address tokenB) external view returns (address pair);
    function allPairs(uint) external view returns (address pair);
    function allPairsLength() external view returns (uint);

    function createPair(address tokenA, address tokenB) external returns (address pair);

    function setFeeTo(address) external;
    function setFeeToSetter(address) external;
    function setMigrator(address) external;
}
```

**Status: Fixed**

The functions have been removed

## DnsClusterMetadataStore

- ### Use SafeMath for all arithmetic operations

**Description:**

There is no check for overflow/underflow in the contract.

**Remediation:**

Use SafeMath library by Openzeppelin for all the arithmetic operations to avoid overflow/underflow in the calculations.

**Status: Fixed**

- ### State variables that could be declared immutable

```
IResourceFeed public resourceFeed;
```

**Description:**

The above constant state variables should be declared immutable to save gas.

**Remediation:**

Add the immutable attributes to state variables that never change after deployment.

**Status**: **Acknowledged by the Auditee**

- ### Code with no use

| Line | Code |
|---|---|
| 25 | `bool isPrivate;` |

**Description:**

This variable is stored in the metadata entry of every cluster. But there has been no actual use found for this variable isPrivate in any of the other contracts.

**Remediation:**

Remove this variable from the struct for metadata or specify a use for it.

**Status**: **Acknowledged by the Auditee**
**Comments from Auditee:** It will be used in later versions.

## ResourceFeed

- **No function ‘getResourcePriceUSDT’ found in contract**

```
    function getResourcePriceUSDT(bytes32 clusterDns, string calldata
name)
        external
        view
        returns (uint256);
```

**Description:**

This function was in the IResourceFeed interface, but there is no such function in the ResourceFeed contract.

**Remediation:**

Remove this function definition.

**Status: Fixed**
This function was removed from the interface.

- **Unused state variables**

```
address public stackToken;
address public USDToken;
```

**Description:**

These state variables are never used in the contract.

**Remediation:**

Remove them.

**Status**: **Acknowledged by the Auditee**
Only the address USDToken was removed.

- ## Use calldata for string parameters to save gas

**Description:**

The following functions use the memory keyword for the string parameter in functions. They can save gas if changed to calldata keyword.

- addResource()
- removeResource()
- setResourceDripRateUSDT()

**Remediation:**

Change memory to calldata.

**Status**: **Acknowledged by the Auditee**

## Oracle (Oracle.sol, IPriceOracle.sol, Staking.sol)

- ### Description Safemath is missing in staking.sol

There is a missing check for overflow/underflow in the contract.

**Remediation**

Use SafeMath library by Openzeppelin as the industry best standard for all the arithmetic operations to avoid overflow/underflow in the calculations.

**Status:** **Acknowledged by the Auditee.**

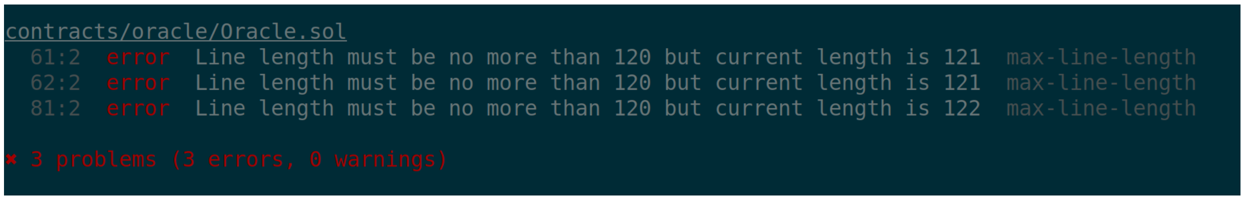
- ### License missing

Developers need to use a license according to your project. The suggestion to use here would be an SPDX license. You can find a list of licenses here: https://spdx.org/licenses/

The SPDX License List is an integral part of the SPDX Specification. The SPDX License List itself is a list of commonly found licenses and exceptions used in free and open or collaborative software, data, hardware, or documentation. The SPDX License List includes a standardized short identifier, the full name, the license text, and a canonical permanent URL for each license and exception.

**Status:** **Acknowledged by the Auditee.**

- ### Linting issue



```
contracts/oracle/Oracle.sol
  61:2  error  Line length must be no more than 120 but current length is 121  max-line-length
  62:2  error  Line length must be no more than 120 but current length is 121  max-line-length
  81:2  error  Line length must be no more than 120 but current length is 122  max-line-length

✖ 3 problems (3 errors, 0 warnings)
```

- **Decimal compatibility for USDT (Oracle.sol)**

As USDT has a different numbers of decimals on different chains, contracts should be selected with caution so as to get the correct rate from USDT to STACK and vice-versa.

**Status: Acknowledged by the Auditee.**

- **Reentrancy Guard missing**

Use the reentrancy OpenZeppelin guard to avoid improper Enforcement of Behavioral Workflow. Link1, Link2

**Status: Acknowledged by the Auditee.**

## Closing Summary:

Overall, smart contracts are very well written and adhere to guidelines.

Numerous issues of high, medium, and low severity were discovered during the initial audit. Most of them are now fixed and checked for correctness.

## Disclaimer: