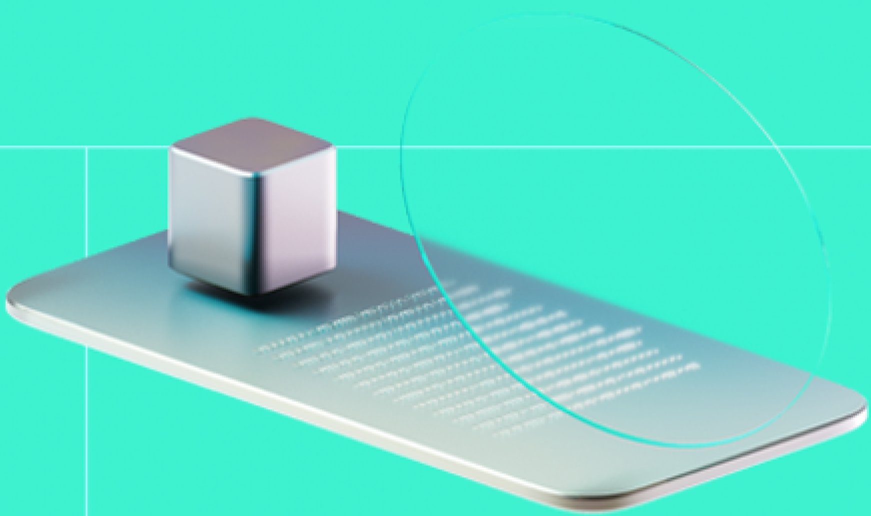
Smart Contract Code Review And Security Analysis Report



Customer: Dexponent

Date: 22/04/2024

We express our gratitude to the Dexponent team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Dexponent emerges as an institutional-grade liquid staking platform designed to cater to institutions' distinct needs. It ensures clean staking practices, employs a non-custodial approach, separates funds, and introduces clETH, enabling instant liquidity for staked ETH. Dexponent also implements Account Abstraction for enhanced security and offers diverse utilities through clETH, presenting a comprehensive solution for institutions entering the liquid staking space.

Platform: EVM

Language: Solidity

Tags: Staking

Timeline: 21/03/2024 - 22/04/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking
Commit	694381d07ab9f2dab336afc54a8bc7e7aa4e42c6

Audit Summary

10/10	8/10	84.26%	10/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9/10

The system users should acknowledge all the risks summed up in the risks section of the report

9	9	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	3
High	1
Medium	5
Low	0

Vulnerability	Status
F-2024-1698 - Amount of Unstaked Tokens Is Not Deducted From The Staked Amount	Fixed
F-2024-1699 - Maximum Deposit Requirement Violation	Fixed
F-2024-1745 - Admin Might 'burn' Tokens From Any Address	Fixed
F-2024-1746 - Liquidating Is Not Possible If The Borrower Do Not Approve Enough Tokens	Fixed
F-2024-1748 - Possible Discrepancy Between The Actual Contract Balance and Recorded Balance	Fixed
F-2024-2120 - Mismatch Between Documentation and Implementation	Fixed
F-2024-2131 - Fund Lock in LoanLogic Contract During liquidateCollateral Process	Fixed
F-2024-2134 - Unfinalized code block	Fixed
F-2024-2136 - Lack Of Validation For The Oracle Data	Fixed

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for Dexponent
Audited By	Max Fedorenko, Roman Tiutiun
Approved By	Grzegorz Trawinski
Website	https://dexponent.com/
Changelog	27/03/2024 - Preliminary Report, 22/04/2024 - Final Report

Table of Contents

System Overview **6**
- Privileged Roles 6

Executive Summary **7**
- Documentation Quality 7
- Code Quality 7
- Test Coverage 7
- Security Score 7
- Summary 7

Risks **8**

Findings **9**
- Vulnerability Details 9
- Observation Details 22
- Disclaimers 34

Appendix 1. Severity Definitions **35**

Appendix 2. Scope **36**

System Overview

Dexponent is a staking protocol with the following contracts:

- ClEth.sol - the contract operates as an ERC20 token but with expanded functionalities tailored for minting, burning, pausing functionalities, managing rewards, and assigning roles to other contracts or addresses.
- TokenProxy.sol - the contract serves as a proxy for interacting with another contract while enabling transparent upgrades.
- WCIETH.sol - the contract is an ERC20 token with additional functionalities inherited from TokenStorage, OwnableUpgradeable, and PausableUpgradeable.
- StakeHolder.sol - the contract serves as a secure holding space, acting as an intermediary or escrow for the Ethereum (ETH) staked by individual users.
- StakingMaster.sol - the contract is the central contract of the staking system. It handles the logic for users staking ETH, managing their stakes, unstake, and interacting with the `CLETH` token and individual `StakeHolder` contracts.
- StakingMasterStorage.sol - the contract defines storage variables and mappings used by a staking master contract.
- Event.sol - contract declares several events used to emit notifications about different actions.
- TokenStorage.sol - abstract contract defines storage variables and emits an event.
- LoanLogic.sol - the contract facilitates the management of loans and collateral within a lending system. It includes functions for creating loans, repaying loans, calculating loan parameters such as interest rates and maximum loan amounts, and liquidating collateral.
- LoanStorage.sol - contract serves as a storage contract holding all state variables and structures related to loans and collateral within a lending system.

Privileged roles

- The owner of the `LoanLogic.sol` contract can update the CLETH price.
- The owner of the `StakeHolder.sol` contract can be deposited to Figment.
- The owner of the `StakingMaster.sol` contract can update the withdrawal status, claim a reward for Wcleth, and claim a reward for Cleth.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

Code quality

The total Code Quality score is **8** out of **10**.

- Code contains redundant code.

Test coverage

Code coverage of the project is **84.26%** (branch coverage).

Security score

Upon auditing, the code was found to contain **3** critical, **1** high, **5** medium, and **0** low severity issues. After the retest, most of the previously identified issues were resolved, leading to a security score **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **9**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- Admin might update the implementation of the Staking and Loan Logic anytime.
- Admin approval is needed to unstake the funds.
- Operations with CLETH token might be paused by admin.
- Admin is responsible for accruing rewards individually to for all the stakers.
- The Price Oracle which is used by the Loan Logic contract is set by the admin and is out of scope.

Findings

Vulnerability Details

F-2024-1746 - Liquidating Is Not Possible If The Borrower Do Not Approve Enough Tokens - Critical

Description:

The `LoanLogic` contract has the `liquidateCollateral` function responsible for the liquidation of the loans. The function might be activated if the loan to value reaches certain threshold `liquidationThreshold`. However the contract tries to get the collateral amount back from the borrower, but the execution of such code requires the borrower allowance.

```
require(clethToken.transferFrom(loan.borrower,
address(this), loan.collateralAmount), "Collateral transfer
failed");
```

This leads to the inability to liquidate the borrowing if the borrower does not want to and lock of the collateral tokens on the contract.

Assets:

- core/base/LoanLogic.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Classification

Impact:

Likelihood [1-5]: 5
Impact [1-5]: 5
Exploitability [0-2]: 1
Complexity [0-2]: 1
Final Score: 4.8 (Critical)
Hacken Calculator Version: 0.6

Severity: Critical

Recommendations

Remediation:

Clarify the expected result and rework the liquidation logic, rework the liquidation logic to enable collateral withdrawals for liquidated borrowings by the admin.

Remediation (Revised Commit: df787f4): The issue was resolved by ensuring that the user's funds are held within the LoanLogic Smart Contract.

Consequently, the condition requiring a clETH fund allowance from the user has been removed.

F-2024-2131 - Fund Lock in LoanLogic Contract During liquidateCollateral Process - Critical

Description:

The `liquidateCollateral()` function of the `LoanLogic.sol` contract, intended for liquidation of collateral for the specified loan.

```
function liquidateCollateral(
uint256 loanId
) public LoanIdNotExits(loanId) nonReentrant {
Loan storage loan = loans[loanId];
require(!loan.isRepaid, "Loan is already repaid or liquidated");
uint256 currentPrice = fetchCLETHPrice();


uint256 loanValueInCLETH = loan.debt / currentPrice;
uint256 currentLTV = calculateMaxLoanAmount(loanValueInCLETH);
require(
currentLTV > liquidationThreshold,
"Loan LTV is below liquidation threshold"
);
transferTokens(clethToken, address(this), loan.collateralAmount);


loan.isRepaid = true;
loan.debt = 0;
totalLoans -= loan.amount;
emit CollateralLiquidated(loanId, loan.collateralAmount);
}
```

In such a case, `liquidateCollateral()` function can be executed by anyone and liquidates any collateral and locks the funds in the `LoanLogic.sol` contract.

Assets:

- core/base/LoanLogic.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Classification

Impact:

Likelihood [1-5]: 5
Impact [1-5]: 5
Exploitability [1-2]: 1
Complexity [0-2]: 1
Final Score: 4.8 (Critical)

Severity: Critical

Recommendations

Remediation:

Conduct a thorough review of the `liquidateCollateral()` function's logic, and create `withdraw()` function.

Remediation (Revised Commit: 079882): The issue was resolved by transferring collateral to a `recoveryAddress`.

Evidences

Proof of Concept (POC) Steps:

Reproduce:

1. **Pre-Condition State**: Assume an user created a loan of using `createLoan()` and has a balance with `clethToken`.
2. **Transaction Execution**: A user attempts to invoke the `liquidateCollateral()` with a specific `loanId`.
3. **Post-Transaction State**: `clethToken` were transferred to the `LoanLogic.sol`.
4. **Post-Transaction State:** The `clethToken` is locked within the contract, preventing any withdrawals by external parties.

```
it("Fund Lock in LoanLogic Contract During liquidateCollateral Process
", async () => {
const balanceBefore = await clETH.balanceOf( loanStorage.address );
console.log( "balanceBefore", balanceBefore );
await loanLogicContract.connect(user1).createLoan(1000)
await loanLogicContract.connect(user2).liquidateCollateral( 1 );
const balanceAfter = await clETH.balanceOf( loanStorage.address );
console.log( "balanceAfter", balanceAfter );
```

Results:

```
balanceBefore 0
balanceAfter 1000
✓ Fund Lock in LoanLogic Contract During liquidateCollateral Process
1 passing (2s)
```

Files:

```
  LoanLogic
    Fund Lock in LoanLogic Contract During liquidateCollateral Process
balanceBefore 0
balanceAfter 1000
      ✓ Fund Lock in LoanLogic Contract During liquidateCollateral Process


  1 passing (2s)
```

F-2024-2134 - Unfinalized code block - Critical

Description: The `StakeHolder.sol` contracts contain functions `sendEth()` logic, this function is used for testing purposes and remains unused within the contracts. The function allows anybody to transfer arbitrary amount of ETH from the contract to any address.

```
function sendEth(
address payable recipient,
uint256 amount
) external payable {
recipient.transfer(amount);
}
```

Assets:

- core/base/StakeHolder.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Classification

Impact: 5/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Critical

Recommendations

Remediation: It is recommended to remove the `sendEth()` function.

Remediation (Revised Commit: 079882): The issue was resolved by removing the `sendEth()` function.

F-2024-1699 - Maximum Deposit Requirement Violation - High

Description:

The project has the functions to validate the maximum amount which is allowed to be deposited within `stake` and `stakeForWCLETH` functions, however the current implementation checks if the amount of tokens to stake is higher than the `MAX_DEPOSIT_AMOUNT` amount, but not lower.

```
function stake() public payable {
require(msg.value >= MAX_DEPOSIT_AMOUNT, "Must send ETH to stake");
// ...
}
```

This lead to the users inability to stake less than 32 ETH.

Assets:

- core/base/StakingMaster.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Classification

Impact:

Likelihood [1-5]: 5
Impact [1-5]: 3
Exploitability [0-2]: 1
Complexity [0-2]: 1
Final Score: 3.8 (High)
Hacken Calculator Version: 0.6

Severity: High

Recommendations

Remediation:

It is recommended to check if the deposited amount is less than 32 ETH, with the updated required statement:

```
require(msg.value <= MAX_DEPOSIT_AMOUNT, "Must send ETH to stake");
```

Remediation (Revised Commit: df787f4): The Dexponent team fixed the issue by implementing `_stake()` function with require check:

```
require(msg.value >= MIN_DEPOSIT_AMOUNT, "Must sent minimum 32 ETH");
```

F-2024-1698 - Amount of Unstaked Tokens Is Not Deducted From The Staked Amount - Medium

Description: The `StakingMaster.sol` contract has the logic within the `unstake()` functions responsible for requesting tokens to be unstacked. The function does not decrease the stake amount which allows users to request unstake amount which is higher than the actual staked amount.

```
function unstake(uint256 amount) public {
require(amount != 0, "Amount can not be zero");
require(StakedBalance[msg.sender] >= amount, "Not enough staked ETH");
require(clETH.balanceOf(msg.sender) >= amount, "Not enough clETH");
clETH.transferFrom(
msg.sender,
address(StakeHolders[msg.sender]),
amount
);
WithdrawalBalance[msg.sender] += amount;
emit Unstaked(msg.sender, amount);
}
```

This allows users to request unstaking tokens more than they staked.

Assets:

- core/base/StakingMaster.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Classification

Impact: Likelihood [1-5]: 5
Impact [1-5]: 1
Exploitability [0-2]: 1
Complexity [0-2]: 1
Final Score: 2.8 (Medium)
Hacken Calculator Version: 0.6

Severity: Medium

Recommendations

Remediation: Update the logic of the unstake function do disallow unstake requests for more than it was previously staked.

Remediation (Revised Commit: df787f4): The Dexponent team resolved the issue by implementing a functionality that verifies if the requested amount to unstake is not greater than the available staked balance, ensuring that users cannot unstake more tokens than they have staked.

F-2024-1745 - Admin Might 'burn' Tokens From Any Address - Medium

Description: The vulnerability in the `burn()` function in the `CLETH.sol` contract allows an admin with the `BURNER_ROLE` to burn tokens from any address. This can result in a significant loss of value for users.

```
function burn(
address from,
uint256 amount
) external onlyRole(BURNER_ROLE) whenNotPaused {
require(amount > 0, "CLETH: burn amount must be greater than zero");
_burn(from, amount);
}
```

Assets:

- core/token/ClEth.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Classification

Impact:

Likelihood [1-5]: 5
Impact [1-5]: 4
Exploitability [1-2]: 2
Complexity [0-2]: 0
Final Score: 2.7 (Medium)

Severity: Medium

Recommendations

Remediation: It is recommended to include function `burnFrom()` from `ERC20BurnableUpgradeable` library from OpenZeppelin. This library includes the `burn()` and `burnFrom()` functions, ensuring that only the owner of `ERC20` tokens or an approved address can burn tokens. By implementing this library, addresses with the `BURNER_ROLE` will no longer have the capability to burn user tokens and tokens will be protected.

Remediation (Revised Commit: df787f4): The Dexponent team resolved the issue by introducing a `burnFrom()` function that includes a validation step to ensure the user's approval before burning tokens from a specific address.

Evidences

Reproduce:

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity ^0.8.18;
```

```
import "forge-std/Test.sol";
import "forge-std/console.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {CLETH} from "../contracts/core/token/ClEth.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1
967Proxy.sol";


contract Hack is Test {
address public alice = makeAddr("bob");
address public bob = makeAddr("bob");
address public owner = makeAddr("owner");


CLETH cleth;


function setUp() public {
uint256 amount = 1000e18;


vm.startPrank(owner);
CLETH impl = new CLETH();
ERC1967Proxy proxy = new ERC1967Proxy(address(impl), "");
cleth = CLETH(address(proxy));
cleth.initialize(owner);
vm.label(address(cleth), "CLETH");


cleth.grantRoles(owner);
cleth.mint(alice, 1000e18);
cleth.mint(bob, amount);
vm.stopPrank();
}


function test_burnTokens() public {
uint256 amountToBurt = 1000e18;


vm.startPrank(owner);
console.log("Alice balance before: ", cleth.balanceOf(alice));
console.log("Bob balance before: ", cleth.balanceOf(bob));


cleth.burn(alice, amountToBurt);
cleth.burn(bob, amountToBurt);


console.log("Alice balance after: ", cleth.balanceOf(alice));
console.log("Bob balance after: ", cleth.balanceOf(bob));
vm.stopPrank();
}
}
```

Files:

```
Running 1 test for test/Hack.t.sol:Hack
[PASS] test_burnTokens() (gas: 44585)
Logs:
  Alice balance before:  2000000000000000000000
  Bob balance before:  2000000000000000000000
  Alice balance after:  0
  Bob balance after:  0

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.00ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

F-2024-1748 - Possible Discrepancy Between The Actual Contract Balance and Recorded Balance - Medium

Description: The `LoanLogic` contract allows to specify the amount of USDC reserves `totalUSDCReserve` and CLETH reserves `totalCLETHReserve` during the initialisation of the smart contract, but these values might be different from the actual one,

This might lead to the contract denial of service until the contract balance is increased to match the recorded values.

Assets:

- core/base/LoanLogic.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Classification

Impact:

Likelihood [1-5]: 3
Impact [1-5]: 3
Exploitability [0-2]: 1
Complexity [0-2]: 1
Final Score: 2.8 (Medium)
Hacken Calculator Version: 0.6

Severity: Medium

Recommendations

Remediation: Verify during the initialisation that the smart contract has enough `usdc` and `cleth` tokens on the balance.

Remediation (Revised Commit: df787f4): The issue was resolved by removing `totalUSDCReserve` and `totalCLETHReserve` during the initialization of the smart contract.

F-2024-2120 - Mismatch Between Documentation and Implementation - Medium

Description:

The Documentation for the **Borrowing Protocol**, specifies certain functionalities and behaviors that do not match the actual implementation of `LoanLogic.sol` contract.

Borrowing Protocol
9. Liquidation of funds:

- The Liquidation threshold is for now a constant value at 92%.

The implementation's `initialize()` function contains:

```
function initialize(
address _usdcTokenAddress,
address _clethTokenAddress,
address _priceFeedAddress
)
public
initializer
...
{
...
liquidationThreshold = 90;
...
}
```

This can result in a potential unexpected delay of asset liquidation where the total value of locked assets is reduced to 90%, instead of the expected 92%, due to a discrepancy between user expectations and actual implementation in code.

- ... liquidation of the loan takes place and `clETH` is transferred to the `clETH` pool on the smart contract.

The implementation's `liquidateCollateral()` function contains:

```
function liquidateCollateral(
uint256 loanId
) public LoanIdNotExits(loanId) nonReentrant {
...
);
transferTokens(clethToken, address(this), loan.collateralAmount);
...
}
```

In this scenario, `clETH` tokens are retained within the contract and are not transferred elsewhere.

Assets:

- core/base/LoanLogic.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Classification

Impact:

Likelihood [1-5]: 5
Impact [1-5]: 1
Exploitability [1-2]: 1
Complexity [0-2]: 1
Final Score: 2.8 (Medium)

Severity: Medium

Recommendations

Remediation: Review the Documentation and update the implementation to match the expected result.

Remediation (Revised Commit: 079882): The issue was resolved by setting the liquidation threshold to 90%, and the `liquidateCollateral()` function in the implementation now transfers tokens to the recovery address.

F-2024-2136 - Lack Of Validation For The Oracle Data - Medium

Description: The `LoanLogic` contract has the `fetchCLETHPrice()` function to fetch the price of USDC/cLETH tokens pair. However the function does not checks if the price is outdated and there is no validation if the return value is zero. The price is used to calculate the collateral amount.

```
function fetchCLETHPrice() public view returns (uint256) {
(
,
/*uint80 roundID*/ int price,
,
/*uint startedAt*/ uint256 timeStamp /*uint80 answeredInRound*/,
) = priceFeed.latestRoundData();

return uint256(price);
}
```

This might lead to the loss of funds due to possible collateral underestimation.

Assets:

- core/base/LoanLogic.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Classification

Impact:

Likelihood [1-5]: 1
Impact [1-5]: 5
Exploitability [0-2]: 1
Complexity [0-2]: 1
Final Score: 3.3 (Critical)
Hacken Calculator Version: 0.6

Severity: Medium

Recommendations

Remediation: Implement the validation to check if the price is outdated and validation if the price if higher than zero. Implement pausing functionality for emergency cases.

Remediation (Revised Commit: 079882): The issue was resolved by verifying whether the price is up-to-date, incorporating a require statement to ensure the price is greater than zero, and pausing functionality was implemented.

Observation Details

F-2024-1683 - Commented Code Part - Info

Description:

In the contract, `LoanStorage.sol` line 36 and `ClEth.sol` lines 17-19, `LoanLogic.sol` line 75 is commented on. The commented code may suggest that the development team disabled part of functionality e.g. for testing purposes and the code might be intended to be used in the final release.

```
ClEth.sol:
// constructor() {
// _disableInitializers();
// }

LoanStorage.sol:
// OwnableUpgradeable.__Ownable_init();

LoanLogic.sol:
// uint256 maxLTV = 70;// Loan-to-Value ratio
```

Assets:

- core/token/ClEth.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Recommendations

Remediation: Remove commented parts of the code.

Remediation (Revised Commit: df787f4): The Dexponent team removed commented parts of the code.

F-2024-1690 - Initializer Is Not Disabled In Constructor - Info

Description: According to the OpenZeppelin documentation, upgradeable contracts should invoke the method `_disableInitializers()` in their `constructor()` to disable implementation contract, preventing them from being used or altered.

However, that functionality is commented in the `ClEth.sol` upgradeable contract.

Assets:

- core/token/ClEth.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Recommendations

Remediation: Follow OpenZeppelin's documentation regarding `_disableInitializers()` in `ClEth.sol` upgradeable contract and uncomment the function.

Remediation (Revised Commit: df787f4): The Dexponent team implemented `constructor` with `_disableInitializers()` in `ClEth.sol`.

F-2024-1693 - Mismatch Between WhitePaper and Implementation - Info

Description:

The WhitePaper for the **StakeHolder Contract**, **CLETH Contract**, and the **StakingMaster Contract** specify certain functionalities and behaviors that do not match the actual implementation of these contracts.

StakeHolder Contract:

1. Constructor:

```
constructor(address _staker, address _masterContract) payable {
require(msg.value > 0, "Must send ETH to create a stake");
staker = _staker;
masterContract = _masterContract;
emit DepositReceived(_staker, msg.value);
}
```

The implementation's `constructor()` contains:

```
constructor(
address _staker,
address _masterContract,
address _masterContractOwner,
IFigmentEth2Depositor _figmentDepositor,
IERC20 _clethToken
) payable {
staker = _staker;
masterContractOwner = _masterContractOwner;
masterContract = _masterContract;
figmentDepositor = _figmentDepositor;
clethToken = _clethToken;
emit DepositReceived(_staker, msg.value);
}
```

2. releaseFunds(uint256 amount):

```
//Function to release funds to the staker
function releaseFunds(uint256 amount) external {
...
}
```

The `releaseFunds()` is missed.

3. deposit():

```
// Function to allow the staker to add more funds to the StakeHolder
function deposit() external payable {
...
}
```

The `deposit()` is missed in the implementation.

CLETH Contract:

3. addReward, setReward:

```
function addReward(address account, uint256 amount) public onlyRole(MI
NTER_ROLE) {
...
}
function setReward(address account, uint256 amount) public onlyRole(MI
NTER_ROLE) {
```

```
...
}
```

`addReward()`, `setReward()` functions are missed in the implementation.

4. claimReward(address account):

```
function claimReward(address account) public {
...
}
```

`claimReward(address account)` is missed in the implementation.

StakingMaster Contract:
1: stake():

```
function stake() public payable {
...
}
```

The `stake()` implementation contains:

```
function stake() public payable {
...
}
```

2: unstake(uint256 amount):

```
function unstake(uint256 amount) public onlyWhitelisted notBlacklisted
{
...
}
```

The `unstake()` implementation contains:

```
function unstake(uint256 amount) public {
...
}
```

3: addToWhitelist(address user) :

```
function addToWhitelist(address user) public onlyOwner {
...
}
```

`addToWhitelist(address user)` is missed in the implementation.

4: addToBlacklist(address user) :

```
function addToBlacklist(address user) public onlyOwner {
...
}
```

`addToBlacklist()` is missed in the implementation.

5: removeFromWhitelist(address user) and removeFromBlacklist(address user) :

```
function removeFromWhitelist(address user) public onlyOwner {
...
}
```

```
function removeFromBlacklist(address user) public onlyOwner {
...
}
```

`removeFromWhitelist()`, `removeFromBlacklist()` are missing in the implementation.

6: transferOwnership(address newOwner) :

```
function transferOwnership(address newOwner) public onlyOwner {
...
}
```

`transferOwnership()` is missed in the implementation.

7: Utility Functions:

```
function getStakedBalance(address account) public view returns (uint25
6) {
...
}

function getLastStakeTime(address account) public view returns (uint25
6) {
...
}

function getTotalPool() public view returns (uint256) {
...
}
function getStakeHolderInfo(address user) public view returns (address
, uint256) {
...
}
```

`getStakedBalance()`, `getLastStakeTime()`, `getTotalPool()`, `getStakeHolderInfo()` are missing in the implementation.

8: depositToNodeOperators:

```
/function setNodeOperatorsDepositor(address _NodeOperatorsDepositor) e
xternal onlyOwner {
NodeOperatorsDepositor = INodeOperatorsEth2Depositor(_NodeOperatorsDep
ositor);
}

function depositToNodeOperators(
address payable stakeHolderAddress,
string calldata pubkey,
string calldata withdrawal_credentials,
string calldata signature,
string calldata deposit_data_root
) external onlyOwner {
...
}

function hexStringToBytes(string memory hexString) internal pure retur
ns (bytes memory) {
...
}

function hexStringToBytes32(string memory source) internal pure return
s (bytes32 result) {
...
}
```

`setNodeOperatorsDepositor()`, `setNodeOperatorsDepositor()`, `hexStringToBytes()`, `hexStringToBytes32()` are missing in the implementation.

However, upon reviewing the actual implementation of the `StakeHolder.sol`, `CLETH.sol` and `StakingMaster.sol` contracts, it is evident that there are different implementations between the WhitePaper and the code.

This discrepancy between the WhitePaper and the contract's code leads to confusion and potential misunderstandings about the contract's behavior and capabilities.

Assets:

- core/base/StakeHolder.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]
- core/token/ClEth.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Recommendations

Remediation: Review the WhitePaper and update the implementation to match the expected result.

Remediation (Revised Commit: df787f4): The Dexponent team provided valid documentation.

F-2024-1702 - Redundant Code Block - Info

Description: The `LoanStorage.sol` and `WClETH.sol` contracts contain functions with logic, but some of these functions include commented, redundant code, or unused parameters.

`__LoanStorage_init()` function in `LoanStorage.sol` contract intended to initialize a contract but has commented code.

```
function __LoanStorage_init() internal initializer {
// OwnableUpgradeable.__Ownable_init();
}
```

`unstake()` function in `WClETH.sol` does not utilize unstake `pubkeys` argument, and redundant function `unpausle()`.

```
function unstake(
uint256 amount,
bytes calldata pubkeys
) public whenNotPaused {
require(amount > 0, "WCLETH: burned amount must be greater than zero")
;
_burn(msg.sender, amount);
emit unstakedRequested(msg.sender, amount, pubkeys);
}

function unpausle() public pure returns (string memory) {
return "HELLo";
}
```

Redundant parts of the code create excessive gas costs.

Assets:

- core/token/WCIETH.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Recommendations

Remediation: Remove redundant code blocks, and parameters in order to consume less gas.

Remediation (Revised Commit: df787f4): The Dexponent team removed unused and redundant code.

F-2024-1751 - Worng Event Naming Convention - Info

Description: The event `withdrawalStatusUpdated` in the `Event.sol` contract does not follow the Solidity naming style. According to Solidity documentation and best practices:

Events in Solidity are typically named using CamelCase starting with an uppercase letter. This convention enhances readability and consistency in codebases.

Assets:

- core/base/events/Event.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Recommendations

Remediation: It is recommended to follow Solidity's best practices and use CameCace for `withdrawalStatusUpdated` event.

Remediation (Revised Commit: df787f4): The Dexponent team implemented the recommendation by using CameCace for `withdrawalStatusUpdated` event.

F-2024-1753 - Missing Events Emitting For Critical Functions - Info

Description: Events for critical state changes should be emitted for tracking actions off-chain. It was observed that events in `LoanLogic.sol` are missing in the following functions:

`updateCLETHPrice()`
`setInterestRateParameters()`
`setLTVParameters()`

Events are crucial for tracking changes on the blockchain, especially for actions that alter significant contract states or permissions. The absence of events in these functions means that external entities, such as user interfaces or off-chain monitoring systems, cannot effectively track these important changes.

Assets:

- core/base/LoanLogic.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Recommendations

Remediation: Consider implementing and emitting events for the necessary functions.

Remediation (Revised Commit: df787f4): The Dexponent team removed `updateCLETHPrice()`, `setInterestRateParameters()`, and `setLTVParameters()`.

F-2024-2121 - Redundant State Variables - Info

Description: The contract `LoanStorage.sol` and `StakeHolder.sol` has redundant state variables and events that are never used within the logic of the contract. Within the contract such events and state variables are redundant: `event FundsSent`, `event ClethReceived`, `clethPrice`, `lastPrice`.

This might indicate unfinalized code, decrease the code readability, and increase Gas expenses during the contract deployment.

Assets:

- core/base/StakeHolder.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Fixed

Recommendations

Remediation: Rework the logic to remove the redundant events and state variable or utilize them.

Remediation (Revised Commit: 079882): The issue was resolved by removing redundant events and state variables: `event FundsSent`, `event ClethReceived`, `clethPrice`, `lastPrice`.

F-2024-2122 - Redundant Math Calculations - Info

Description: In the `LoanLogic.sol` contract, the `calculateMaxLTV()` function contains redundant math during the computation of the `currentUtilization` variable:

```
uint256 currentUtilization = ((totalLoans / 1e18) * 100) /
((totalfund > 0 ? (totalfund / 1e18) : 1));
```

Assets:

- core/base/LoanLogic.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Accepted

Recommendations

Remediation: The code block should be replaced with the equivalent to:

```
uint256 currentUtilization = (totalLoans * 100) /
(totalfund > 0 ? totalfund : 1);
```

to improve gas usage and code readability. This change simplifies the calculation and makes the code more straightforward to understand.

Remediation (Revised Commit: f23c156): The issue is unfixed, redundant calculations are still present within the code.

F-2024-2920 - Admin Can Initiate clETH Token Minting - Info

Description: During the initialization of the clETH token contract, the system administrator is assigned the `DEFAULT_ADMIN_ROLE`, which acts as the default admin role for all other roles.

This enables the admin to grant the `MINTER_ROLE` to any account, allowing it to mint clETH tokens.

Assets:

- core/token/ClEth.sol [https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking]

Status: Accepted

Recommendations

Remediation: Assign the `stakingMaster` contract as the role manager for the `MINTER_ROLE` during initialization. This ensures that no one else can reassign the `MINTER_ROLE`.

Resolution: The client is aware of the risk and is responsible for proper role management.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Primary Scope

Details

Repository	https://gitlab.ardourlabs.com/dexponent/smart-contracts/staking
Commit	694381d07ab9f2dab336afc54a8bc7e7aa4e42c6
Whitepaper	https://docs.dexponent.com/
Requirements	https://docs.google.com/document/d/1d2KDcb8vZUns6Pm-xYeguJrxaUb6__kZGSdwTYbuXew/edit#heading=h.r6ltp1yoax1g
Technical Requirements	https://docs.google.com/document/d/1d2KDcb8vZUns6Pm-xYeguJrxaUb6__kZGSdwTYbuXew/edit#heading=h.r6ltp1yoax1g

Contracts in Scope

./contracts/core/ClEth.sol

./contracts/core/TokenProxy.sol

./contracts/core/Proxy.sol

./contracts/core/WCIETH.sol

./contracts/core/base/StakeHolder.sol

./contracts/core/base/StakingMaster.sol

./contracts/core/base/StakingMasterStorage.sol

./contracts/core/base/events/Event.sol

./contracts/core/base/storage/TokenStorage.sol

./contracts/core/base/loanLogic.sol

./contracts/core/base/loanStorage.sol