CONFIDENTIAL

CYBERSECURITY AUDIT REPORT

Version v1.1

This document details the process and results of the smart contract audit performed by CyStack from 18/12/2023 to 29/12/2023.

Audited for

Oquity Foundation Pte. Ltd.

Audited by

Vietnam CyStack Joint Stock Company

Contents

Disclaimer

Smart Contract Audit only provides findings and recommendations for an exact commitment of a smart contract codebase. The results, hence, are not guaranteed to be accurate outside of the commitment, or after any changes or modifications made to the codebase. The evaluation result does not guarantee the nonexistence of any further findings of security issues.

Time-limited engagements do not allow for a comprehensive evaluation of all security controls, so this audit does not give any warranties on finding all possible security issues of the given smart contract(s). CyStack prioritized the assessment to identify the weakest security controls an attacker would exploit. We recommend Oquity Foundation Pte. Ltd. conduct similar assessments on an annual basis by internal, third-party assessors, or a public bug bounty program to ensure the security of smart contract(s).

This security audit should never be used as investment advice.

Version History

Version	Date	Release notes
1.0	29/12/2023	The first report was sent to the client.
1.1	04/01/2024	The only finding was rejected and omitted from the report.

Contact Information

Company	Representative	Position	Email address
Oquity Foundation Pte. Ltd.	Edric Collins	CEO	contact@oquity.finance
CyStack	Nguyen Ngoc Anh	Sales Manager	anhntn@cystack.net

Auditors

Fullname	Role	Email address
Nguyen Huu Trung	Head of Security	trungnh@cystack.net
Nguyen Ba Anh Tuan	Auditor	
Vu Trong Khoi	Auditor	
Ha Minh Chau	Auditor	
Nguyen Van Huy	Auditor	

Introduction

From 18/12/2023 to 29/12/2023, Oquity Foundation Pte. Ltd. engaged CyStack to evaluate the security posture of the Oquity Finance Lending Platform of their contract system. Our findings and recommendations are detailed here in this initial report.

1.1 Audit Details

Audit Target

In this audit project, CyStack focused only on certain smart contracts belonging to the Oquity Finance Lending Platform of Oquity Foundation Pte. Ltd..

The basic information of Oquity's smart contracts is as follows:

Item	Description
Project Name	Oquity Finance Lending Platform (GitHub: oquitydev/oquity_contract)
Issuer	Oquity Foundation Pte. Ltd.
Website	https://oquity.finance/
Platform	N/A
Language	Solidity
Codebase	https://github.com/oquitydev/oquity_contract/commit/764187a566967a32cdc700a2e2b583a327fbe65e
Commit	764187a566967a32cdc700a2e2b583a327fbe65e
Audit method	Whitebox

Oquity is a decentralized lending protocol that allows interest-free loans by collateralizing ONUS. This protocol is characterized by decentralization, non-asset retention, and efficiency.

Loans are disbursed in VNDO, a base currency pegged to VND, and must maintain a minimum collateral ratio of 130%. In addition to collateral, the loans are backed by a Stable Liquidity Pool filled with VNDO, along with a general guarantee from borrowers acting as Stabilizing Liquidity Providers.

Oquity lending platform is characterized by the features listed below:

- Borrow VNDO collateral with ONUS through Loan origination.
- Enhance the stability of the Oquity system by providing VNDO to the Stabilization Liquidity Pool in exchange for rewards.
- Stake OQTY to earn fee revenue from borrowing or converting VNDO.
- Convert VNDO to an equivalent amount of ONUS when the anchor price of VNDO is reduced.

The following contracts are audited by CyStack:

1. The contract **Manager.sol** is an abstract contract that implements the **IManager.sol** interface and inherits from the *Ownable* contract. This contract contains a boolean variable called *enable* to monitor the operating state of the system and a modifier called *systemEnable* to ensure that the system is active when executing critical functions. The *enableSystem* function is used to enable or disable the system, only the owner has the right to perform this operation. When the state of the system changes, a *SystemEnableHandle* event is emitted for notification. This contract provides a mechanism to manage the operating state of the system and requires owner confirmation before changing this state.
2. The contract **PriceFeed.sol**, whose logic is currently defined in the contract **PriceFeedTestnet.sol**, stores the current price in the *_price* variable and manages the list of price updaters through the mapping updater. Only those who are confirmed as updaters have the right to update prices. The *getPrice* function returns the current price without changing the data in the contract. The *fetchPrice* function performs the same functionality as *getPrice* and is part of the **IPriceFeed.sol** interface. The *setPrice* function can only be called by updaters and is used to update the price. When the price is updated, a *PriceUpdated* event is emitted to notify of the change. Contract owners can add or remove updaters via the *addUpdater* function. This contract provides a simple and secure mechanism to manage and provide prices for the system.
3. The contract **StabilityPool.sol** manages the amount of LUSD and Ether deposited into the system by users and interacts with other contracts to perform functions related to borrowing, trove management, OQUSD, and other operations. This contract plays an important role in controlling access rights, using modifier functions to ensure safety. It also handles events and notifications through events to record important contract changes, while maintaining snapshots to record account status and key values. These functions synchronize to create a stable and flexible system.
4. **CommuniyIssuance.sol** is the contract in charge of distributing OQTY rewards to users, with a secret transfer mechanism based on whitelisting.
5. **OQUSD.sol** is an ERC20 contract, representing a USD stable coin for Oquity and also the main currency on OQTY. This token is implemented with a mechanism to block token transfers into OQuity's core contracts so that users do not mistakenly deposit OQUSD into these contracts.
6. **OQTYToken.sol** is an ERC20 contract that defines the secondary currency for OQuity. Fees generated by the system and rewards for early adopters are calculated and paid via this currency.
7. **OQTYStaking.sol** is used to manage OQTY's staking.

The privilege system of Oquity includes two special roles: Owner and Updater.

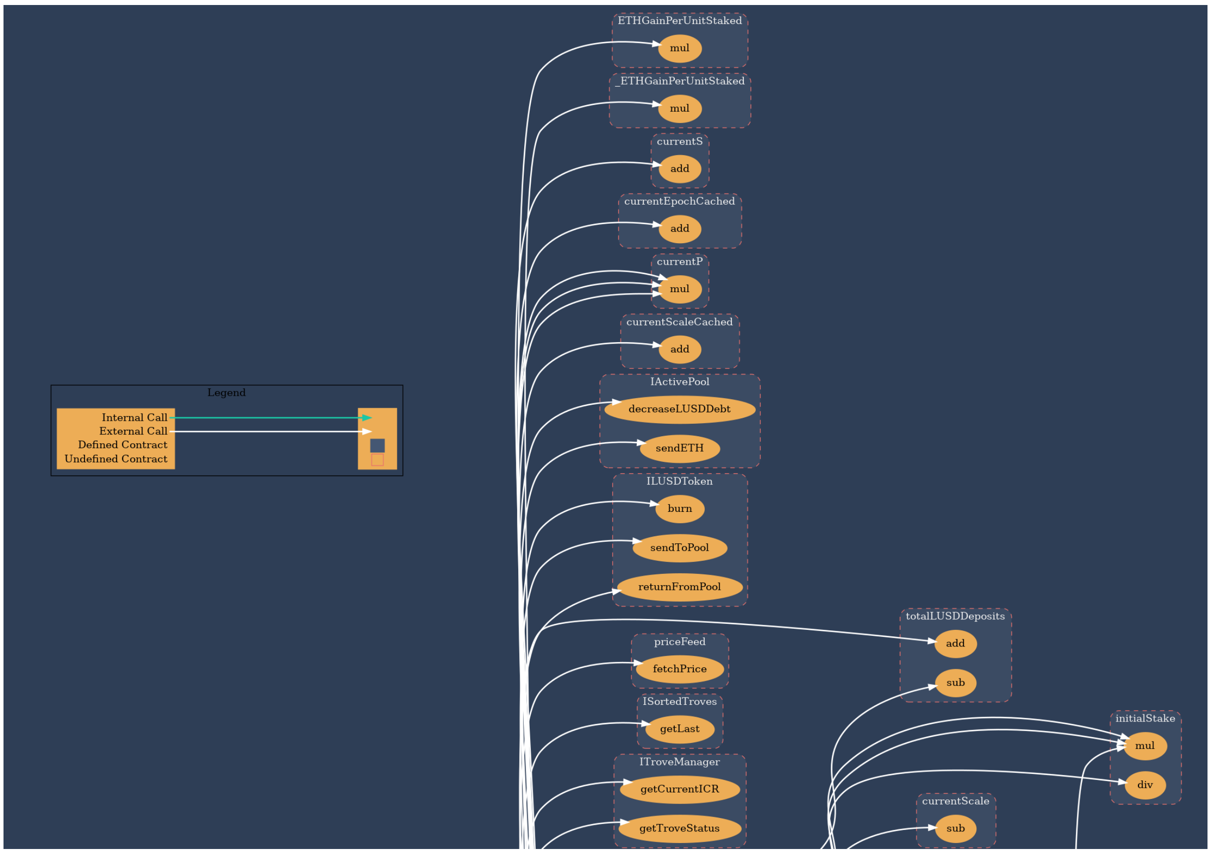
Only Owners can perform the following functions in StabilityPool.sol:

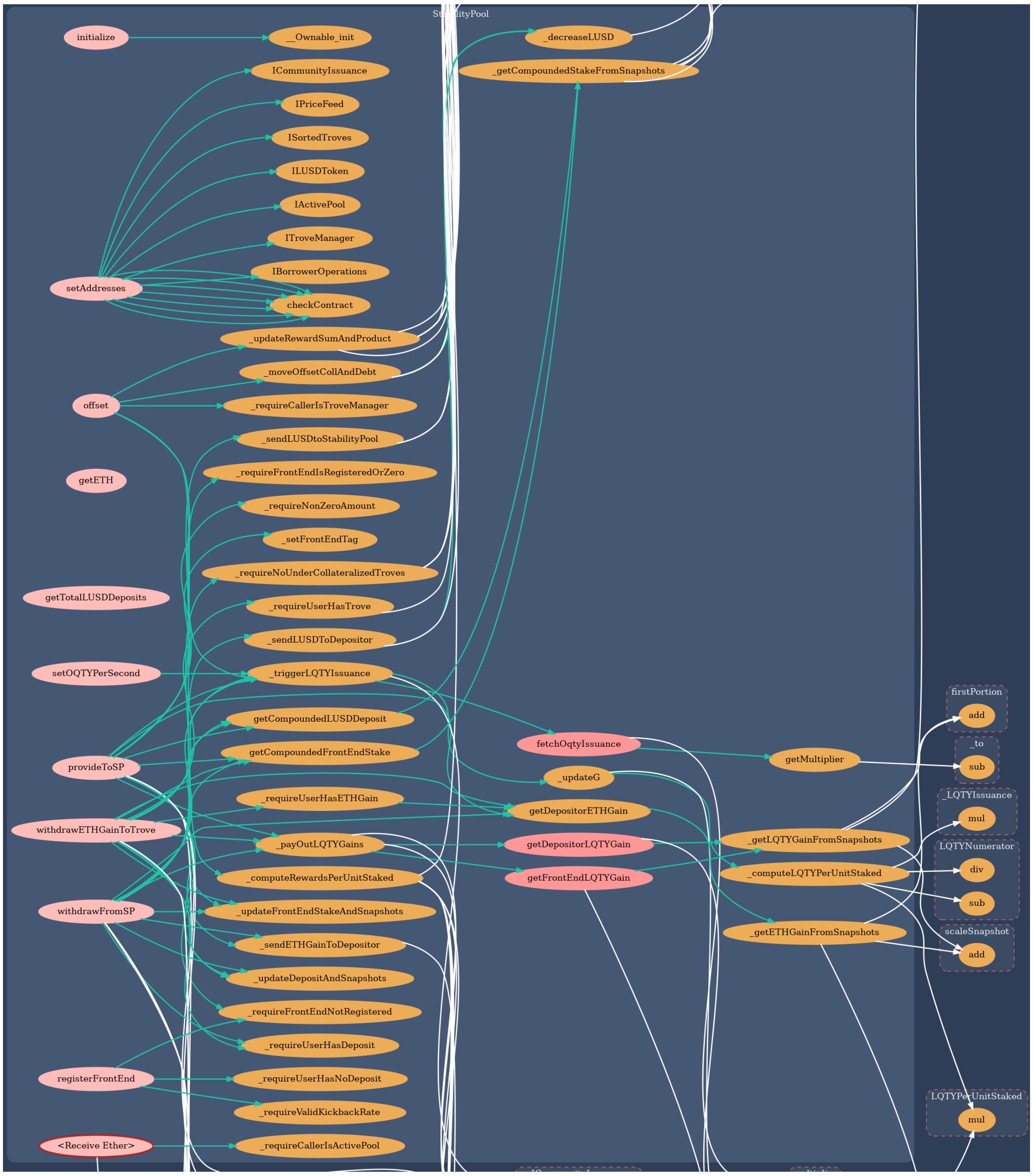
- setAddresses: set addresses for _borrowerOperationsAddress, _troveManagerAddress, _activePoolAddress, _lusdTokenAddress, _sortedTrovesAddress, _priceFeedAddress, _communityIssuanceAddress
- setOQTYPerSecond: set value for _oqtyPerSecond, only uint256 values are accepted

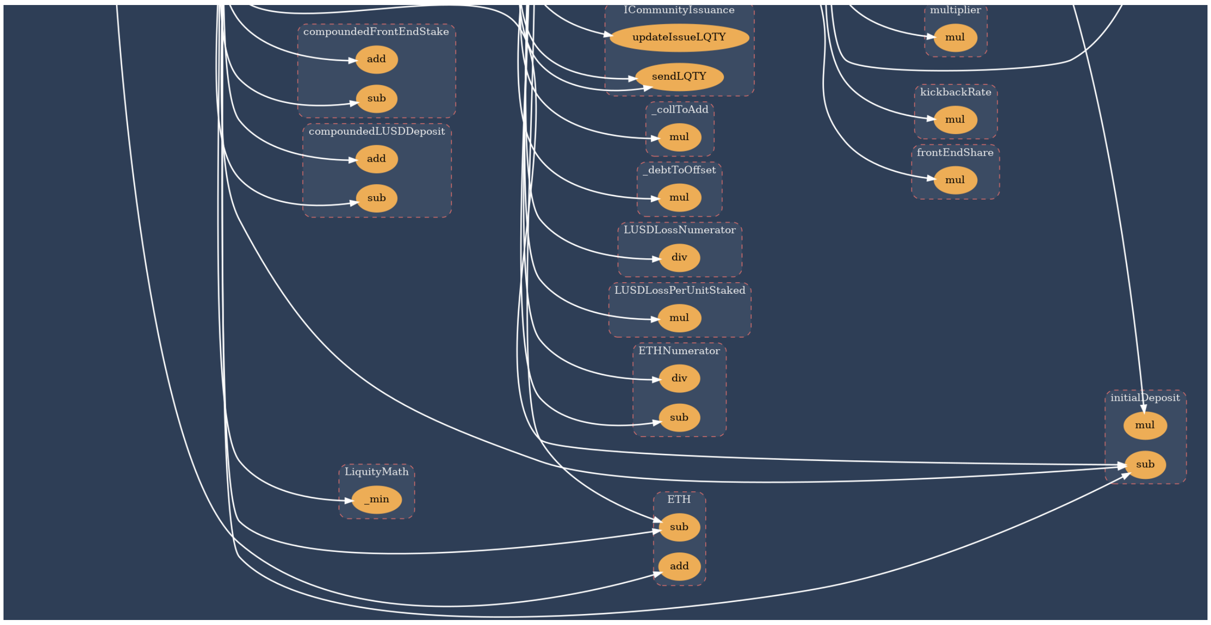
Both Owners and Updaters can perform the following functions in PriceFeedTestnet.sol:

- addUpdater: escalate a user to the role updater
- setPrice: update the price

The function calls in most contracts are illustrated in the following graphs:




StabilityPool
initialize
__Ownable_init
ICommunityIssuance
IPriceFeed
ISortedTroves
ILUSDToken
IActivePool
ITroveManager
IBorrowerOperations
setAddresses
checkContract
_updateRewardSumAndProduct
_moveOffsetCollAndDebt
offset
_requireCallerIsTroveManager
_sendLUSDtoStabilityPool
getETH
_requireFrontEndIsRegisteredOrZero
_requireNonZeroAmount
_setFrontEndTag
_requireNoUnderCollateralizedTroves
getTotalLUSDDeposits
_requireUserHasTrove
_sendLUSDToDepositor
setOQTYPerSecond
_triggerLQTYIssuance
getCompoundedLUSDDeposit
getCompoundedFrontEndStake
provideToSP
_requireUserHasETHGain
withdrawETHGainToTrove
_payOutLQTYGains
_computeRewardsPerUnitStaked
withdrawFromSP
_updateFrontEndStakeAndSnapshots
_sendETHGainToDepositor
_updateDepositAndSnapshots
_requireFrontEndNotRegistered
_requireUserHasDeposit
registerFrontEnd
_requireUserHasNoDeposit
_requireValidKickbackRate
<Receive Ether>
_requireCallerIsActivePool
_decreaseLUSD
_getCompoundedStakeFromSnapshots
fetchOqtyIssuance
getMultiplier
_updateG
getDepositorETHGain
getDepositorLQTYGain
getFrontEndLQTYGain
_getLQTYGainFromSnapshots
_computeLQTYPerUnitStaked
_getETHGainFromSnapshots
firstPortion
add
_to
sub
_LQTYIssuance
mul
LQTYNumerator
div
sub
scaleSnapshot
add
LQTYPerUnitStaked
mul



Audit Service Provider

CyStack is a leading security company in Vietnam with the goal of building the next generation of cybersecurity solutions to protect businesses against threats from the Internet. CyStack is a member of Vietnam Information Security Association (VNISA) and Vietnam Alliance for Cybersecurity Products Development.

CyStack's researchers are known as regular speakers at well-known cybersecurity conferences such as BlackHat USA, BlackHat Asia, Xcon, T2FI, etc. and are talented bug hunters who discovered critical vulnerabilities in global products and are acknowledged by their vendors.

1.2 Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient and working according to its specifications. The audit activities can be grouped in the following three categories:

1. **Security:** Identifying security related issues within each contract and within the system of contracts.
2. **Sound Architecture:** Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. **Code Correctness and Quality:** A full review of the contract source code. The primary areas of focus include:
   - Correctness
   - Readability
   - Sections of code with high complexity
   - Improving scalability
   - Quantity and quality of test coverage

1.3 Audit Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology:

- **Likelihood** represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- **Impact** measures the technical loss and business damage of a successful attack;
- **Severity** demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: High, Medium and Low, i.e., H, M and L respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., Critical, Major, Medium, Minor and Informational (Info) as the table below:

Impact	*High*	Critical	Major	Medium
	Medium	Major	Medium	Minor
	Low	Medium	Minor	Informational
		High	*Medium*	*Low*
			Likelihood	

CyStack firstly analyses the smart contract with open-source and also our own security assessment tools to identify basic bugs related to general smart contracts. These tools include Slither, securify, Mythril, Sūrya, Solgraph, Truffle, Geth, Ganache, Mist, Metamask, solhint, mythx, etc. Then, our security specialists will verify the tool results manually, make a description and decide the severity for each of them.

After that, we go through a checklist of possible issues that could not be detected with automatic tools, conduct test cases for each and indicate the severity level for the results. If no issues are found after manual analysis, the contract can be considered safe within the test case. Else, if any issues are found, we might further deploy contracts on our private testnet and run tests to confirm the findings. We would additionally build a PoC to demonstrate the possibility of exploitation, if required or necessary.

The standard checklist, which applies for every SCA, strictly follows CyStack's Smart Contract Weakness Classification. This classification is built, strictly following the Smart Contract Weakness Classification Registry (SWC Registry), and is updated frequently according to the most recent emerging threats and new exploit techniques. The checklist of testing according to this classification is shown in Appendix C.

In general, the auditing process focuses on detecting and verifying the existence of the following issues:

- **Data Issues**: Finding bugs in data processing, such as improper names and labels for variables, incorrect inheritance orders and unsafe calculations.
- **Description Issues**: Checking for improper controls of user input that leads to malicious output rendering.
- **Environment Issues**: Inspecting errors related to the environment of some specific Solidity versions.
- **Interaction Issues**: Reviewing the interaction of different smart contracts to locate bugs in handling external calls and controlling the balance and flows of token transfers.
- **Interface Issues**: Investigating the misusage of low-level and token interfaces.
- **Logic Issues**: Testing the code logic and error handlings in the smart contract code base, such as self-DoS attacks, verifying strong randomness, etc.
- **Performance Issues**: Identifying the occurrence of improper byte padding, unused functions and other issues that leads to high gas consumption.
- **Security Issues**: Finding common security issues of the smart contract(s), for example integer overflows, insufficient verification of authenticity, improper use of cryptographic signature, etc.
- **Standard Issues**: Focusing on identifying coding bugs related to general smart contract coding conventions and practices.

The final report will be sent to the smart contract issuer with an executive summary for overview and detailed results for acts of remediation.

1.4 Audit Scope

Assessment	Target	Type
Original target (commit: 764187a566967a32cdc700a2e2b583a327fbe65e)		
White-box testing	IManager.sol	Solidity code file
White-box testing	Manager.sol	Solidity code file
White-box testing	IPriceAggregator.sol	Solidity code file
White-box testing	PriceFeed.sol	Solidity code file
White-box testing	PriceFeedTestnet.sol	Solidity code file
White-box testing	StabilityPool.sol	Solidity code file
White-box testing	CommunityIssuance.sol	Solidity code file
White-box testing	IOQTYToken.sol	Solidity code file
White-box testing	OQTYToken.sol	Solidity code file
White-box testing	OQTYStaking.sol	Solidity code file
White-box testing	OQUSDToken.sol	Solidity code file

Executive Summary

Security issues by severity

No vulnerabilities were found.

Security issues by categories

No vulnerabilities were found.

Table of security issues

No vulnerabilities were found.

Recommendations

Based on the results of this smart contract audit, CyStack has the following high-level key recommendations:

Key recommendations	
Issues	CyStack conducted security audit for different contracts in Oquity Finance Lending Platform. A Minor issue related to controlling address input was found, but it was not considered a vulnerability and was rejected by the Oquity Foundation Pte. Ltd.. It was an intended design since the input values for addresses were validated on the application layer. In conclusion, Oquity Foundation Pte. Ltd. maintains a good security posture without any vulnerabilities on their smart contracts.
Recommendations	CyStack recommends Oquity Foundation Pte. Ltd. to evaluate the audit results with several different security audit third parties for the most accurate conclusion.
References	• https://consensys.github.io/smart-contract-best-practices/known_attacks • https://consensys.github.io/smart-contract-best-practices/recommendations/ • https://medium.com/@knownsec404team/ethereum-smart-contract-audit-checklist-ba9d1159b901

Conclusion

CyStack had conducted a security audit for Oquity Foundation Pte. Ltd.'s smart contracts. An issue was found, but it was the intended design and already had a value control mechanism on the application layer, which is not in the audit scope. Overall, the audited smart contracts have included the best practices for smart contract development and have passed our security assessment for smart contracts.

To improve the quality of this report, and for CyStack's Smart Contract Audit report in general, we greatly appreciate any constructive feedback or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

Appendices

Appendix A – Security Issue Status Definitions

Status	Definition
Open	The issue has been reported and currently being review by the smart contract developers/issuer.
Unresolved	The issue is acknowledged and planned to be addressed in future. At the time of the corresponding report version, the issue has not been fixed.
Resolved	The issue is acknowledged and has been fully fixed by the smart contract developers/issuer.
Rejected	The issue is considered to have no security implications or to make only little security impacts, so it is not planned to be addressed and won't be fixed.

Appendix B – Severity Explanation

Severity	Definition
CRITICAL	Issues, considered as critical, are straightforwardly exploitable bugs and security vulnerabilities. It is advised to immediately resolve these issues in order to prevent major problems or a full failure during contract system operation.
MAJOR	Major issues are bugs and vulnerabilities, which cannot be exploited directly without certain conditions. It is advised to patch the codebase of the smart contract as soon as possible, since these issues, with a high degree of probability, can cause certain problems for operation of the smart contract or severe security impacts on the system in some way.
MEDIUM	In terms of medium issues, bugs and vulnerabilities exist but cannot be exploited without extra steps such as social engineering. It is advised to form a plan of action and patch after high-priority issues have been resolved.
MINOR	Minor issues are generally objective in nature but do not represent actual bugs or security problems. It is advised to address these issues, unless there is a clear reason not to.
INFO	Issues, regarded as informational (info), possibly relate to "guides for the best practices" or "readability". Generally, these issues are not actual bugs or vulnerabilities. It is recommended to address these issues, if it makes effective and secure improvements to the smart contract codebase.

Appendix C – Smart Contract Weakness Classification

ID	Name	Description
	Data Issues	
SLD-101	Initialization	Check for Interger Division, Interger Overflow and Underflow, Interger Sign, Interger Truncation and Wrong Operator
SLD-102	Calculation	Check for State Variable Default Visibility, Hidden Built-in Symbols, Hidden State Variables and Incorrect Inheritance Order
SLD-103	Hidden Weaknesses	Check for Unintialized Local/State Variables and Unintialized Storage Variables
	Description Issues	
SLD-201	Output Rendering	Check for RightToLeft Override Control Character
	Environment Issues	
SLD-302	Supporting Software	Check for Deletion of Dynamic Array Elements and Usage of continue Statements In do-while -statements
	Interaction Issues	
SLD-401	Contract Call	Check for Delegatecall to Untrusted Callee, Re-entrancy and Unhandled Exception
SLD-402	Ether Flow	Check for Unprotected Ether Withdrawal, Unexpected Ether Balance, Locked Ether and Pre-sent Ether
	Interface Issues	
SLD-501	Parameter	Check for Externally Controlled Call/delegatecall Data/Address, Hash Collisions with Multiple Variable Length Arguments, Short Address Attack and Signature with Wrong Parameter
SLD-502	Token Interface	Check for Non-standard Token Interface
	Logic Issues	
SLD-601	Assembly Code	Check for Arbitrary Jump with Function Type Variable, Returning Results Using Assembly Code in Constructor and Specifying Function Variable as Any Type

SLD-602	Denial of Service (DoS)	Check for potential DoS due to failed call, by complex fallback function, by gaslimit and by non-exsistent address or malicious contracts
SLD-603	Fairness Problems	Check for Weak Sources of Randomness from Chain Attributes, Usage of Block Values as A Proxy for Time, Results of Contract Execution Affected by Miners and Transaction Order Dependence
SLD-604	Storage	Check for Storage Overlap Attack
SLD-605	Bussiness Logic	Check for Business Logic errors in code
	Performance Issues	
SLD-701	Gas Consumption	Check for Gas Griefing, Byte Padding, Invariants in Loop and Invariants for State Variables that are not declared constant
	Security Issues	
SLD-801	Authority Control	Check for Write to Arbitrary Storage Location, Replay Attack, Suicide Contract, Usage of tx.origin for Authentication/Authorization, Wasteful Contract and Wrong Constructor Name
SLD-802	Privacy	Check for Lack of Proper Signature Verification, Signature Malleability, Non-public Variables that are accessed by public/external functions and Public Data
	Standard Issues	
SLD-901	Maintainability	Check for Implicit Visibility, Non-standard Naming, The Use of Too Many Digits, Unlimited/Outdated Compiler Versions and Usage of Deprecated Built-in Symbols
SLD-902	Programming	Check for Code with No Effect, Message Call with Hardcoded Gas Amount, Presence of Unused Variables, View/Constant Functions that change contract states and Improper Usage of require, assert or revert