CyStack

CONFIDENTIAL

CYBERSECURITY AUDIT REPORT

Version v1.1

This document details the process and results of the smart contract audit performed by CyStack from 02/11/2023 to 10/11/2023.

Audited for

Vay Network Services Private Limited

Audited by

Vietnam CyStack Joint Stock Company

© 2023 CyStack. All rights reserved.
Portions of this document and the templates used in its production are the property of CyStack and cannot be copied (in full or in part) without CyStack's permission.
While precautions have been taken in the preparation of this document, CyStack the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of CyStack's services does not guarantee the security of a system, or that computer intrusions will not occur.

Contents

1 Introduction **4**
1.1 Audit Details 4
1.2 Audit Goals 6
1.3 Audit Methodology 6
1.4 Audit Scope 8

2 Executive Summary **9**

3 Detailed Results **11**

4 Conclusion **21**

5 Appendices **22**
Appendix A – Security Issue Status Definitions 22
Appendix B – Severity Explanation 23
Appendix C – Smart Contract Weakness Classification 24

© 2023 CyStack. All rights reserved.

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 Vay Network Services Private Limited conducting 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 an investment advice.

Version History

Version	Date	Release notes
1.0	11/11/2023	The first report was sent to the client. Most findings were unresolved.
1.1	24/11/2023	All accepted findings were resolved.

© 2023 CyStack. All rights reserved.

Contact Information

Company	Representative	Position	Email address
Vay Network Services Private Limited	Vineet Mago	VP	vineet@vayana.com
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	

© 2023 CyStack. All rights reserved.

Introduction

From 02/11/2023 to 10/11/2023, Vay Network Services Private Limited engaged CyStack to evaluate the security posture of the VDP Digital Assets 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 on smart contracts belongs to the VDP Digital Assets Lending Platform of Vay Network Services Private Limited.

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

Item	Description
Project Name	VDP Digital Assets Lending Platform (Bitbucket: vayana/vdp-contracts-shared)
Issuer	Vay Network Services Private Limited
Website	https://vayana.com/
Platform	N/A
Language	Solidity
Codebase	https://bitbucket.org/vayana/vdp-contracts-shared/src/700c6e513f01f907182a427ab0fb54db3128445f/contracts/
Commit	700c6e513f01f907182a427ab0fb54db3128445f
Audit method	Whitebox

© 2023 CyStack. All rights reserved.

VDP Digital Assets Lending Platform is constructed with the following main contracts:

1. *Libraries* contracts: These are contracts that support main functional contracts in the system. **Accountant.sol** helps monitor and manage loan states and payments schedules. **Validation.sol** provides validation methods. **Utils.sol** defines actual arithmetic financial calculations for the lending system.
2. **AccessController.sol**, **AccessControlVerifier.sol**, **AccessControlVerifierPausable.sol**, **GlobalConfig.sol** and **ProviderAccessManager.sol**: These contracts provide access control and the associated action to each role in the system. Roles are classified into Admin, Upgrader, Platform Admin and Provider.
3. **LoanRegistry.sol** manages and monitors loan information of each borrower.
4. **Migration.sol** is an utility contract that records successful migration.
5. **ParticipantRegistry.sol** is a contract for Provider and is implemented to keep track of participant ID through ERC721 tokens.
6. **TermLoanAccount.sol** is designed to manage individual term loans within a decentralized system. It facilitates the lifecycle of a loan, from activation and drawdown to repayment and potential default or write-off. The contract is part of a larger system that involves multiple contracts working together.
7. **TermLoanActivations.sol** manages the activation and deployment of term loans in a decentralized system. The contract facilitates the creation, update, acceptance, and rejection of loan activation requests. It works in conjunction with other contracts and interfaces to deploy corresponding loan accounts and vaults.
8. **TermLoanVault.sol** serves as a smart contract managing the funding and withdrawal processes for a decentralized lending system. The contract handles contributions from investors, tracks treasury deposits, and facilitates the withdrawal of funds.
9. **WriteOff.sol** helps decide to forgive or cancel unpaid loans through votes.

© 2023 CyStack. All rights reserved.

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 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:

© 2023 CyStack. All rights reserved.

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

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.

© 2023 CyStack. All rights reserved.

- **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.

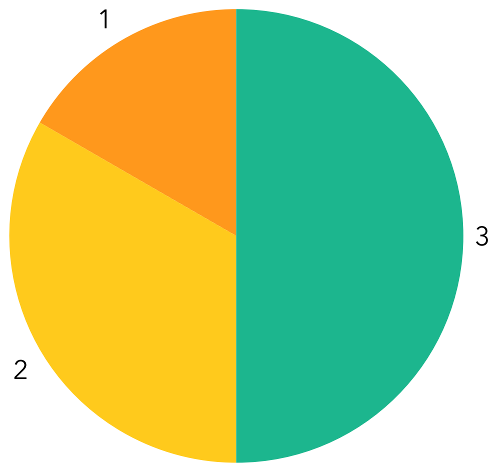
1.4 Audit Scope

Assessment	Target	Type
Original target (commit: 700c6e513f01f907182a427ab0fb54db3128445f)		
White-box testing	/src/contracts/Interfaces/ *.sol	Solidity code files
White-box testing	/src/contracts/Libraries/ *.sol	Solidity code files
White-box testing	/src/contracts/*.sol	Solidity code files
Re-test target (commit: acdcb7ede64682091c6f874f5e393ecbac984337)		
White-box testing	/src/contracts/Interfaces/ *.sol	Solidity code files
White-box testing	/src/contracts/Libraries/ *.sol	Solidity code files
White-box testing	/src/contracts/*.sol	Solidity code files

© 2023 CyStack. All rights reserved.

Executive Summary

Security issues by severity



Legend

- Critical
- Major
- Medium
- Minor
- Info

Security issues by categories

Category	Count
Denial of Service (DoS) (SLD-602)	1
Business Logic (SLD-605)	2
Gas Consumption (SLD-701)	1
Maintainability (SLD-901)	1
Programming (SLD-902)	1

Table of security issues

ID	Status	Vulnerability	Severity
#vayana-006	Resolved	Potential Denial of Service by block gas limit	MEDIUM
#vayana-001	Resolved	Missing zero address validation	MINOR
#vayana-002	Resolved	Floating pragma	INFO
#vayana-005	Resolved	Boolean equality	INFO
#vayana-003	Rejected	Missing input validations in setter functions	MINOR
#vayana-004	Rejected	Gas optimization with error statements	INFO

© 2023 CyStack. All rights reserved.

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 VDP Digital Assets Lending Platform. No issues with severity higher than Medium had been found. A total of six issues were found, related to logic errors, coding conventions and performance. Currently, all four accepted findings are unresolved.
Recommendations	CyStack recommends Vay Network Services Private Limited 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

© 2023 CyStack. All rights reserved.

Detailed Results

1. Potential Denial of Service by block gas limit

Issue ID	#vayana-006
Category	Logic Issues - Denial of Service (DoS)
Description	Every transaction in Ethereum requires 21000 gas on top of the computations made in the contract. An Ethereum block has a maximum limit of 30 million gas. Exceeding this limit will cause the transaction to be reverted. If not properly managed, this can render certain functions of the contract inoperable. Either the array grows over time, or a malicious actor can accumulate a large number of loans and facilitate a DoS attack.
Severity	MEDIUM
Location(s)	LoanRegistry.sol: 33, 246, 248, 253, 261, 264-265, 268-269
Status	Resolved
Remediation	Either impose a limit on array size, or avoid having large arrays that grow over time and looping across the entire data structure. If the above is not feasible, plan for operations over such arrays to spread over multiple blocks, and therefore require multiple transactions.

Description

borrowerLoans is a mapping from an uint256 (borrower's id) to an array of ***Loan*** structure.

```
…
33      mapping(uint256 => Loan[]) public borrowerLoans; // Tracking all loans for particular borrower
…
```


© 2023 CyStack. All rights reserved.

There's no limit on the ***Loan*** array's size, nor does any operation on ***borrowerLoans*** have batch processing mechanism implemented in case the size of ***Loan*** array grows too large:

```
…
118     function markNPA(Loan calldata loan) external onlyRole(LOAN_ADMIN_ROLE, accessManager) {
119         require(IActivations(loan.loanActivationsAddress).isValidLoan(loan), "Invalid loan");
120         require(IActivations(loan.loanActivationsAddress).checkNPA(loan), "Loan does not meet NPA
        ↪   conditions");
121         uint256 borrower = IActivations(loan.loanActivationsAddress).getBorrower(loan);
122
123         uint256 length = borrowerLoans[borrower].length;
124         for (uint16 i; i < length; i++) {
125             ITermLoanActivations(loan.loanActivationsAddress).markNPA(borrowerLoans[borrower][i]);
126         }
127
128         _pauseBorrower(borrower);
129     }
…
```

```
…
245     function _unmarkNPAAll(address activation, uint256 borrower) internal {
246         uint256 length = borrowerLoans[borrower].length;
247         for (uint256 i; i < length; i++) {
248             if (ITermLoanActivations(activation).checkNPA(borrowerLoans[borrower][i])) {
249                 return;
250             }
251         }
252         for (uint256 i; i < length; i++) {
253             ITermLoanActivations(activation).unmarkNPA(borrowerLoans[borrower][i]);
254         }
255         if (!hasNPAOrDefault(borrower)) {
256             _unpauseBorrower(borrower);
257         }
258     }
…
```

```
…
260     function hasNPAOrDefault(uint256 borrower) internal view returns (bool) {
261         uint256 length = borrowerLoans[borrower].length;
262         for (uint256 i; i < length; i++) {
263             if (
264                 IActivations(borrowerLoans[borrower][i].loanActivationsAddress).getLoanStatus(
265                     borrowerLoans[borrower][i]
266                 ) ==
267                 IAccount.Status.NPA ||
268                 IActivations(borrowerLoans[borrower][i].loanActivationsAddress).getLoanStatus(
269                     borrowerLoans[borrower][i]
270                 ) ==
271                 IAccount.Status.Default
272             ) {
273                 return true;
274             }
275         }
276         return false;
277     }
…
```


© 2023 CyStack. All rights reserved.

2. Missing zero address validation

Issue ID	#vayana-001
Category	Logic Issues - Business Logic
Description	The function changeFactory is lack of zero address check for newFactory, which may cause unexpected results.
Severity	MINOR
Location(s)	TermLoanActivations.sol: 313-317
Status	Resolved
Remediation	Add check of zero address before using newFactory in any operation.

Description

The codelines where the issue occurs:

```
…
313     function changeFactory(bool factoryType, address newFactory) external
        ↪ onlyRole(UPGRADER_ROLE, accessManager) {
314         if (factoryType) {
315             vaultFactory = newFactory;
316         } else accountFactory = newFactory;
317     }
…
```

The codebase can be improved as following:

```
…
920     function changeFactory(bool factoryType, address newFactory) external
        ↪ onlyRole(UPGRADER_ROLE, accessManager) {
921         require(newFactory != address(0), "newFactory cannot be address(0)");
922         if (factoryType) {
923             vaultFactory = newFactory;
924         } else accountFactory = newFactory;
925     }
926
```


© 2023 CyStack. All rights reserved.

3. Floating pragma

Issue ID	#vayana-002
Category	Standard Issues - Maintainability
Description	Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.
Severity	INFO
Location(s)	Interfaces/IVault.sol: 2
Status	Resolved
Remediation	It is recommended to use a fixed pragma version, as future compiler versions may handle certain language constructions in a way the developer did not foresee. Using a floating pragma may introduce several vulnerabilities if compiled with an older version.

Description

The codeline where floating pragma is used:

```
…
2  pragma solidity ^0.8.17;
…
```

The code can be revised as written below:

```
…
2  pragma solidity 0.8.17; // or any version according to the version requirements from
   ↪  libraries, the latest (0.8.23) is preferable
…
```


© 2023 CyStack. All rights reserved.

4. Boolean equality

Issue ID	#vayana-005
Category	Standard Issues - Programming
Description	Boolean constants can be used directly in conditionals like if and else statements. In several contracts, some conditionals are set with comparisons between a boolean constant and the value true (or false).
Severity	INFO
Location(s)	TermLoanAccount.sol: 182, 225, 249, 297 TermLoanActivations.sol: 303, 439 TermLoanVault.sol: 189, 256, 458
Status	Resolved
Remediation	It is recommended to use boolean constants directly.

Description

The codelines where these issues occur:

TermLoanAccount.sol

```
…
182         require(externalIdUsed[externalId] == false, "External ID already used");
…
225         require(externalIdUsed[externalId] == false, "External ID already used");
…
249         require(externalIdUsed[externalId] == false, "External ID already used");
…
297         require(externalIdUsed[externalId] == false, "External ID already used");
…
```

TermLoanActivations.sol

```
…
303         require(externalIdUsed[externalId] == false, "External ID already used");
…
439         require(externalIdUsed[externalId] == false, "External ID already used");
…
```

TermLoanVault.sol

```
…
189         require(externalIdUsed[externalId] == false, "External ID already used");
…
256         require(externalIdUsed[externalId] == false, "External ID already used");
…
458         require(externalIdUsed[externalId] == false, "External ID already used");
…
```


© 2023 CyStack. All rights reserved.

The issue can be resolved as follows:

TermLoanAccount.sol

```
…
182         require(!externalIdUsed[externalId], "External ID already used");
…
225         require(!externalIdUsed[externalId], "External ID already used");
…
249         require(!externalIdUsed[externalId], "External ID already used");
…
297         require(!externalIdUsed[externalId], "External ID already used");
…
```

TermLoanActivations.sol

```
…
303         require(!externalIdUsed[externalId], "External ID already used");
…
439         require(!externalIdUsed[externalId], "External ID already used");
…
```

TermLoanVault.sol

```
…
189         require(!externalIdUsed[externalId], "External ID already used");
…
256         require(!externalIdUsed[externalId], "External ID already used");
…
458         require(!externalIdUsed[externalId], "External ID already used");
…
```


© 2023 CyStack. All rights reserved.

5. Missing input validations in setter functions

Issue ID	#vayana-003
Category	Logic Issues - Business Logic
Description	Values passed into setter functions by msg.sender should always be validated to ensure that no unexpected behaviours might occur. Without validation, an arbitrary user may manipulate contract data or execute functions with malicious intent.
Severity	MINOR
Location(s)	Libraries/Accountant.sol: 420, 448
Status	Rejected
Remediation	Always add a check for values from msg.sender in setter functions to ensure that the changes in smart contract data are legitimate. It is recommended to have these functions reviewed by professionals and covered by unit tests.

Description

The codelines where input validations are missing:

```
 …
415     function setLoanVariables(
416         ITermLoanAccount.ComputationVariables memory loanState,
417         ITermLoanActivations.LoanManagement memory loanVars,
418         uint64 loanCreationTimestamp
419     ) public pure returns (ITermLoanAccount.ComputationVariables memory) {
420         loanState.pendingPrincipalPayments = loanVars.noOfPrincipalPayments;
 …
437     }
 …

 …
439     function setRecalledState(
440         ITermLoanAccount.ComputationVariables memory loanState,
441         ITermLoanActivations.LoanManagement memory loanVars,
442         uint64 loanCreationTimestamp
443     ) external view returns (ITermLoanAccount.ComputationVariables memory) {
 …
445         uint256 principal = loanState.outstandingPrincipal;
446         (, uint256 interestDue, uint256 overduePenalty, uint256 remWaiveOff) = getTotalPendingAmount(
447             loanState,
448             loanVars,
449             currTime,
450             loanCreationTimestamp
451         );
 …
464     }
 …
```


© 2023 CyStack. All rights reserved.

The issue can be resolved by adding require statements before the operations:

```
…
415     function setLoanVariables(
416         ITermLoanAccount.ComputationVariables memory loanState,
417         ITermLoanActivations.LoanManagement memory loanVars,
418         uint64 loanCreationTimestamp
419     ) public pure returns (ITermLoanAccount.ComputationVariables memory) {
420         require(loanVars.noOfPrincipalPayments > 0, "Invalid value for loanVars.noOfPrincipalPayments");
421         loanState.pendingPrincipalPayments = loanVars.noOfPrincipalPayments;
…
439     }
…
```

```
…
439     function setRecalledState(
440         ITermLoanAccount.ComputationVariables memory loanState,
441         ITermLoanActivations.LoanManagement memory loanVars,
442         uint64 loanCreationTimestamp
443     ) external view returns (ITermLoanAccount.ComputationVariables memory) {
…
445         require(loanState.outstandingPrincipal > 0, "Invalid value for loanState.outstandingPrincipal")
446         uint256 principal = loanState.outstandingPrincipal;
447         (, uint256 interestDue, uint256 overduePenalty, uint256 remWaiveOff) = getTotalPendingAmount(
448             loanState,
449             loanVars,
450             currTime,
451             loanCreationTimestamp
452         );
…
465     }
…
```

Updated on re-test targets

The functions setLoanVariables/setRecalledState were renamed to computeLoanVariables/computeRecalledState as these are pure/view functions but had misleading names.

© 2023 CyStack. All rights reserved.

6. Gas optimization with error statements

Issue ID	#vayana-004
Category	Performance Issues - Gas Consumption
Description	Revert() statements are used in the contracts listed below. Since Solidity v0.8.4, custom errors have been introduced as better alternatives to revert statements. Developers can pass custom errors with dynamic data while reverting the transaction, in addition to making the whole implementation a bit cheaper than using revert.
Severity	INFO
Location(s)	Libraries/Accountant.sol: 51, 176 TermLoanAccount.sol: 189 TermLoanVault.sol: 171
Status	Rejected
Remediation	It is recommended to replace revert statements with error statements to save gas.

Description

The codelines where these issues occur:

Libraries/Accountant.sol

```
…
51              isClosed ? revert("Amount exceeds foreclosure amount") : (payment, installmentFees, isClosed)
                ↪  = prePayment(
52                  amount,
53                  payment,
54                  loanVars,
55                  paymentId
56              );
…
…           if (payment.outstandingPrincipal < amount) revert("Amount exceeds foreclosure amount");
177
```

TermLoanAccount.sol

```
…
187         if (penaltyType == WaiveOff.Overdue) loanState.overdueWaiveOff = value;
188         else if (penaltyType == WaiveOff.Prepayment) loanState.prepaymentWaiveOff = value;
189         else revert("Invalid Penalty Type");
…
```


© 2023 CyStack. All rights reserved.

TermLoanVault.sol

```
…
164         if (currentSupply == capacity) {
165             require(_isWithinDrawdownPeriod(), "Cannot make drawdown after drawdown period");
166             _drawdown(activations.getBorrower(activationId));
167          } else if (currentSupply >= (capacity * drawdownThreshold) / 10000) {
168             require(!_isWithinFundingPeriod(), "Cannot make partial drawdown during funding period");
169             require(_isWithinDrawdownPeriod(), "Cannot make partial drawdown after drawdown period");
170             _drawdown(activations.getBorrower(activationId));
171          } else revert("Insufficient supply");
…
```

Use error() instead of revert() to optimize gas consumption:

Libraries/Accountant.sol

```
…
51              isClosed ? revert("Amount exceeds foreclosure amount") : (payment, installmentFees, isClosed)
                ↪   = prePayment(
52                  amount,
53                  payment,
54                  loanVars,
55                  paymentId
56              );
…
…            if (payment.outstandingPrincipal < amount) error("Amount exceeds foreclosure amount");
177
```

TermLoanAccount.sol

```
…
187          if (penaltyType == WaiveOff.Overdue) loanState.overdueWaiveOff = value;
188          else if (penaltyType == WaiveOff.Prepayment) loanState.prepaymentWaiveOff = value;
189          else error("Invalid Penalty Type");
…
```

TermLoanVault.sol

```
…
164         if (currentSupply == capacity) {
165             require(_isWithinDrawdownPeriod(), "Cannot make drawdown after drawdown period");
166             _drawdown(activations.getBorrower(activationId));
167          } else if (currentSupply >= (capacity * drawdownThreshold) / 10000) {
168             require(!_isWithinFundingPeriod(), "Cannot make partial drawdown during funding period");
169             require(_isWithinDrawdownPeriod(), "Cannot make partial drawdown after drawdown period");
170             _drawdown(activations.getBorrower(activationId));
171          } else error("Insufficient supply");
…
```

Updated on re-test targets

No changes were made since the remediation for this finding will cause a major impact on UI and other dependent components.

© 2023 CyStack. All rights reserved.

Conclusion

CyStack had conducted a security audit for Vay Network Services Private Limited's smart contracts. A total of six issues were found, but none of these issues represented critical bugs or security problems. Four of these issues then were accepted by the Vay Network Services Private Limited team. After a re-test on the new codebase for Vay Network Services Private Limited's smart contracts, CyStack confirmed that all found issues were resolved. No new issues were found for the additional functions in the smart contracts. 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.

© 2023 CyStack. All rights reserved.

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.

© 2023 CyStack. All rights reserved.

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.

© 2023 CyStack. All rights reserved.

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

© 2023 CyStack. All rights reserved.

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

© 2023 CyStack. All rights reserved.