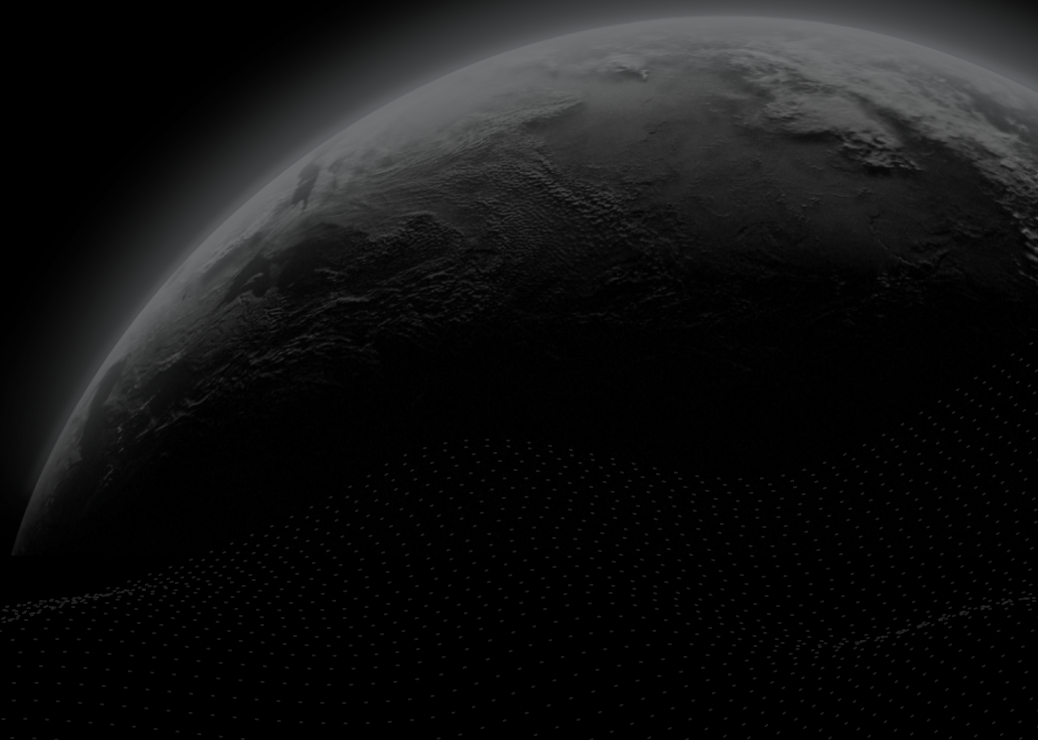
CERTIK

Security Assessment

Venus - Isolated Pools

CertiK Assessed on Jun 19th, 2023



CertiK Assessed on Jun 19th, 2023

Venus - Isolated Pools

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
DeFi	Binance Smart Chain (BSC)	Formal Verification, Manual Review, Static Analysis

LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 06/19/2023	N/A

CODEBASE

https://github.com/VenusProtocol/isolated-pools

View All in Codebase Page

COMMITS

base: 1ec61863d0d498a16c3fb4da2dc15021b06c96a6

update1: bdb7ee099eb9d29784d359846b6c44bfbb0a68c4

update2: bdb7ee099eb9d29784d359846b6c44bfbb0a68c4

View All in Codebase Page

Vulnerability Summary

36	30	2	0	4	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined

1	Critical	1 Resolved	Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
9	Major	7 Resolved, 2 Mitigated	Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
4	Medium	4 Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
8	Minor	7 Resolved, 1 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
14	Informational	11 Resolved, 3 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | VENUS - ISOLATED POOLS

Summary

- Executive Summary
- Vulnerability Summary
- Codebase
- Audit Scope
- Approach & Methods

Review Notes

- System Overview
- Contract Summaries
  - PoolRegistry
  - Factories
  - Risk Fund
  - Shortfall
  - Rewards
  - AccessControlManager
  - PoolLens
  - Rate Models
  - VToken
  - Comptroller

Dependencies

- Third Party Dependencies
  - Recommendation
  - Alleviation
- Out Of Scope Dependencies
  - Recommendations
  - Alleviation

Findings

- CVP-01 : Improper Checks For `collateralFactor` and `liquidationThreshold`
- COM-01 : Use of `_ensureMaxLoops()` May Lock Users' Positions
- CVP-02 : Lack of Validation On `LiquidationOrder[]`
- CVP-04 : Possible Insufficient Checks in `liquidateAccount()` and `healAccount()`

GLOBAL-01 : Centralization Related Risks

GLOBAL-02 : Centralized Control of Contract Upgrade

RFF-02 : `_swapAsset()` Returns Wrong Value If Path Has Length Greater Than 2

VPB-01 : Function `transferReserveForAuction()` May Transfer Reserves To Wrong Contract

VPB-23 : Potential Rounding Error Exploit

VTV-01 : Function `reduceReserves()` is Unprotected

COP-01 : Case When Collateral Equals Scaled Borrows Is No Longer Covered

VPB-04 : Interfaces Not Inherited And Improper/Missing Implementation

VPB-05 : Lack of Storage Gap in Upgradeable Contract

VTV-02 : `mintBehalf()` Can Mint Tokens to the Zero Address

COP-05 : Price Is Not Updated For All Users Markets In `preBorrowHook()`

PLV-01 : Incorrect Calculation In `PoolLens`

RFR-01 : Ineffective Check For `minAmountToConvert`

VPB-06 : Missing Zero Address Validation

VPB-08 : Changing `convertibleBaseAsset` Can Cause Issues

VPB-10 : `initialize()` Is Unprotected And Inconsistent Init Function

VPH-01 : Missing Input Validation

VTV-03 : Approval Race Condition

COP-02 : Possible Gas Issue With `updatePrices()`

CVP-05 : Storage Variables Declared Outside of `ComptrollerV1Storage`

JRM-01 : Unnecessary Inheritance

RDR-01 : Anyone Can Claim Rewards For Another User

RFR-02 : Duplicate Imports

SSP-01 : Inconsistent Comment And Code

VPB-02 : Incompatibility With Fee on Transfer Tokens

VPB-11 : Volatile Conditions

VPB-12 : Missing or Unused Emit Events

VPB-13 : Missing Access Restriction

VPB-19 : Atypical Constructor Implementation

VPB-21 : Access Control Changes

VPH-02 : Typos and Inconsistencies

VTV-04 : Unnecessary Reference to `VTokenInterface`

Optimizations

COP-04 : Unnecessary And Incorrect Check

PRV-04 : Unnecessary Parameter

RDR-02 : Missing/Unnecessary Functionality

VPB-15 : Unused State Variable

VPB-16 : Unneccessary Use of Payable Casting

VPB-22 : User-Defined Getters

VTV-05 : Inefficient Memory/Storage Parameters

WPM-01 : Variables That Could Be Declared as Immutable

Formal Verification

Considered Functions And Scope

Verification Results

Appendix

Disclaimer

CODEBASE | VENUS - ISOLATED POOLS

Repository

https://github.com/VenusProtocol/isolated-pools

Commit

base: 1ec61863d0d498a16c3fb4da2dc15021b06c96a6
update1: bdb7ee099eb9d29784d359846b6c44bfbb0a68c4
update2: bdb7ee099eb9d29784d359846b6c44bfbb0a68c4
update3: 1c8cc30e72686c525b2f68ec5e2b493f1405823c
update4: 8d5a0e261ad056a37def2707b2df02c06e31bd4e
Addendum: 78c5a0947dda78e3df87ad16b6f2a4521ee12968
update5: d09f33129e113876351f4005d544a012480de41b
update6: f075e8256a5215d438ff610f34cd3e25eea7c79d
update7: 74d3d384225da763e6fcecc688e0f06576016b33
update8: 108aa561f07115b8a571bfba92c89f6055b675c7

AUDIT SCOPE | VENUS - ISOLATED POOLS

82 files audited ● 4 files with Acknowledged findings ● 21 files with Resolved findings ● 57 files without findings

ID	Repo	Commit	File	SHA256 Checksum
● RDR	VenusProtocol/isolated-pools	1ec6186	contracts/Rewards/RewardsDistributor.sol	4a9b2f282db027b773dc8fdcbd897ebe9fc07cda10ec735adec128f4b165417f
● RFR	VenusProtocol/isolated-pools	1ec6186	contracts/RiskFund/RiskFund.sol	d1eb4abddd326ae7b47c81006c2e86be8623d65780233502fd32488c46388290
● SSV	VenusProtocol/isolated-pools	1ec6186	contracts/Shortfall/Shortfall.sol	13d8db841ba260d3f24076e81a05672589f116c424516b5acc8f9a19f39edd55
● VTV	VenusProtocol/isolated-pools	1ec6186	contracts/VToken.sol	1e6c0336b7eb8a30522e64a6d1563550973165dd02e6fc78844ffb84e06dce9c
● VTP	VenusProtocol/isolated-pools	1ec6186	contracts/Factories/VTokenProxyFactory.sol	e6ece35cb13ced140bd8067b4b33eb4622c51f6b36350dfe59cdcfc3cef06529
● PLL	VenusProtocol/isolated-pools	1ec6186	contracts/Lens/PoolLens.sol	9edb4e0f80d48926890caf49bd33337f7b88e28d337b538f470bcc84f1086232
● PRP	VenusProtocol/isolated-pools	1ec6186	contracts/Pool/PoolRegistry.sol	7bb9668e81c219a18b834d109c608c95f12a3f54d49eacf041aba27d32852c55
● PRI	VenusProtocol/isolated-pools	1ec6186	contracts/Pool/PoolRegistryInterface.sol	544ebab3e5018092966c2bd8e98133e04d7bc00ee0af443335507ce138d5d37b
● PSR	VenusProtocol/isolated-pools	1ec6186	contracts/RiskFund/ProtocolShareReserve.sol	0923dff0af375650d88e045f7ae42ddb176595635c910faaf0370f75fe79a499
● RHR	VenusProtocol/isolated-pools	1ec6186	contracts/RiskFund/ReserveHelpers.sol	00ba651e947cca121833f3a7191b289455246f44a4f12986e29a06eeebdd551d
● CVP	VenusProtocol/isolated-pools	1ec6186	contracts/Comptroller.sol	f4016b8865bdd6cb4fc1cf7480ff1135cbbe2d606c544026d7214a1e51b3a534

ID	Repo	Commit	File	SHA256 Checksum
CIV	VenusProtocol/isolated-pools	1ec6186	contracts/ComptrollerInterface.sol	da3cb627e7492ee10a88e5137d3d9dbddaeaefa5e93cf1ec370e96da6af7d25f
CSV	VenusProtocol/isolated-pools	1ec6186	contracts/ComptrollerStorage.sol	790d33ace9ce6f2328ebc6e6c531bb2dfe9962562f8913f0e8ea96c024e2b325
JRM	VenusProtocol/isolated-pools	1ec6186	contracts/JumpRateModelV2.sol	0c230d1cd2f9892c8bb412391ebb503d9feac998f4d9f635eac1d566316d3f9b
ACV	VenusProtocol/isolated-pools	78c5a09	contracts/Governance/AccessControlled.sol	1b3dcf671a82b395cc2fe3a07dd45bbe53e688b4ce6b131b1d180cafe0c962f4
PLV	VenusProtocol/isolated-pools	78c5a09	contracts/Lens/PoolLens.sol	08677cae74275e8706b06dc94fb87fbe7b531ca89d65c091a2ac3f9b7f13a214
PRV	VenusProtocol/isolated-pools	78c5a09	contracts/Pool/PoolRegistry.sol	d9b600d1c56ade60c2cf6bdf55a126533531cd4cf1c707a6ce14037c99d8143c
PSF	VenusProtocol/isolated-pools	78c5a09	contracts/RiskFund/ProtocolShareReserve.sol	07e8c05e7243ae958a4df338016ab6e797ce0c633a4f2b9c0d3f8b4b9fb1d9d1
RFF	VenusProtocol/isolated-pools	78c5a09	contracts/RiskFund/RiskFund.sol	1296b5dfc63a62c9ea634da32d9988d16c40b8708975df62d3266c527695d552
SSP	VenusProtocol/isolated-pools	78c5a09	contracts/Shortfall/Shortfall.sol	cb59a7a27d3800f9cd490384f3bf2ac45cce9fd984cacae6a66c9789efb6ab5f
COM	VenusProtocol/isolated-pools	78c5a09	contracts/Comptroller.sol	20e35b1e45e7744b03117e9322098bbbc88140c8d632f9daa5952fec7122fcf4
CSP	VenusProtocol/isolated-pools	78c5a09	contracts/ComptrollerStorage.sol	5a5b672cb386f73de6bb35ad09fdaafe6f30b3bf5124e23ea10fa0e3d82f6ad6
MLL	VenusProtocol/isolated-pools	78c5a09	contracts/MaxLoopsLimitHelper.sol	c7489773b250997392c9df9e844ad93eb26c6d629319e26263782a27e7e098c1
VVP	VenusProtocol/isolated-pools	78c5a09	contracts/VToken.sol	45725fc77f3b9667f496b674fa1fc1b39ffcf66a739a5598cf81542add70a45d
WPM	VenusProtocol/isolated-pools	78c5a09	contracts/WhitePaperInterestRateModel.sol	16ebb1dc364715cbba77c09899ade04e660c0bc07e29a6abf15e3e8c7e3ac89c

ID	Repo	Commit	File	SHA256 Checksum
JRF	VenusProtocol/isolated-pools	1ec6186	contracts/Factories/JumpRateModelFactory.sol	30cb416a9951cfc1adc28044a745bae8d04e4cc550ee4615cd678c9c20e1fa85
WPR	VenusProtocol/isolated-pools	1ec6186	contracts/Factories/WhitePaperInterestRateModelFactory.sol	46b606fa19478ca8925f0710b7853a877a61f404506d0f9857eea6d5c0a6ab6d
ACM	VenusProtocol/isolated-pools	1ec6186	contracts/Governance/AccessControlManager.sol	7afbfacadeefe296caf6d685a7338cc6b288c5a45ac8a8a492fbfdacc82788b9
UBP	VenusProtocol/isolated-pools	1ec6186	contracts/Proxy/UpgradeableBeacon.sol	4223872d108aa2fb6e46bc82dc6d854b696d881cf558bb5a41d7c805496f632d
IPS	VenusProtocol/isolated-pools	1ec6186	contracts/RiskFund/IProtocolShareReserve.sol	d1cb8573a681d8c4559eb0d49c09d5aaa78b8367f0d01e6d85c238ff5a6205f1
IRF	VenusProtocol/isolated-pools	1ec6186	contracts/RiskFund/IRiskFund.sol	e803a5be43c6c7b077ad32fc75733ae9f201edfd722eaa886471df37bcf6078d
BJR	VenusProtocol/isolated-pools	1ec6186	contracts/BaseJumpRateModelV2.sol	68502438c28cefb4899ebf918243e53c8bf357c4aafcf3fa7b8467b225dea713
ERV	VenusProtocol/isolated-pools	1ec6186	contracts/ErrorReporter.sol	79f36a1ed754fa29a67b9cba7af7f9366f6a15b1a454d2d46757ea4f563e6008
ENE	VenusProtocol/isolated-pools	1ec6186	contracts/ExponentialNoError.sol	c28a36694c1bd6672407d92d0f292f6412db71ab7714837d7e1d1d1fdf1c60a3
IPV	VenusProtocol/isolated-pools	1ec6186	contracts/IPancakeswapV2Router.sol	34f70ef95045f307bb76c59aab82293b5f2cfcfd92e6e83b4ef7104dfdf1ba17
IRM	VenusProtocol/isolated-pools	1ec6186	contracts/InterestRateModel.sol	943d50bdda3bbf79ff4547fc133e73fa0fb5b8f0ce252e088d8203959490441c
VTI	VenusProtocol/isolated-pools	1ec6186	contracts/VTokenInterfaces.sol	8f075ceaaa7cae8a5e0626ab36800bc05d6f5ccf3c3d55417c3f97d95ca328e5

ID	Repo	Commit	File	SHA256 Checksum
WPI	VenusProtocol/isolated-pools	1ec6186	contracts/WhitePaperInterestRateModel.sol	c1119b03bb18061837188a10f5b09b13a2fb3655bb92bcb34ae848f1b314705c
JRP	VenusProtocol/isolated-pools	78c5a09	contracts/Factories/JumpRateModelFactory.sol	5a23eefaf558195abc6d59c58679db0cb4ae5c66ee68306185c8ce32a460a8c7
VTF	VenusProtocol/isolated-pools	78c5a09	contracts/Factories/VTokenProxyFactory.sol	1e343e748e719c04c58b4641ee3169dd0d6a0f593e3aa96636ba7943da70fc9e
WPF	VenusProtocol/isolated-pools	78c5a09	contracts/Factories/WhitePaperInterestRateModelFactory.sol	46b606fa19478ca8925f0710b7853a877a61f404506d0f9857eea6d5c0a6ab6d
ACG	VenusProtocol/isolated-pools	78c5a09	contracts/Governance/AccessControlManager.sol	06ff8f240e201847e75b3ae2a0919d247211cbbea2e7b57ff3450f178bd6f031
IAC	VenusProtocol/isolated-pools	78c5a09	contracts/Governance/IAccessControlManager.sol	21fe9f4e35f6717c4256ee60fcc16f70cacb38e7e90502d73b534d457318e915
PIP	VenusProtocol/isolated-pools	78c5a09	contracts/Pool/PoolRegistryInterface.sol	e1beb12fcf5543fea1c921064c04ba930446bb1b1e7e766e385325e3082a2fe3
UBV	VenusProtocol/isolated-pools	78c5a09	contracts/Proxy/UpgradeableBeacon.sol	0dbd882da05e5181f31dc7c7e481f402f46bb9f7a04e5b07ee1d5fac3f96b235
RDV	VenusProtocol/isolated-pools	78c5a09	contracts/Rewards/RewardsDistributor.sol	d9d37e17cdc5806f1664160bdab11fbff6f4e2a6c3e68003e86f217318848ae1
IPF	VenusProtocol/isolated-pools	78c5a09	contracts/RiskFund/IProtocolShareReserve.sol	d1cb8573a681d8c4559eb0d49c09d5aaa78b8367f0d01e6d85c238ff5a6205f1
IRR	VenusProtocol/isolated-pools	78c5a09	contracts/RiskFund/IRiskFund.sol	4f42d8a328742eff673a7e8e6e74f92d72f035880684c6665f059420a97bb641

ID	Repo	Commit	File	SHA256 Checksum
RHF	VenusProtocol/isolated-pools	78c5a09	contracts/RiskFund/ReserveHelpers.sol	55dd79bd0565b85806fef1ef5e8c7cda8cfbcfe74f516dd4826f0be9fab18c17
ISS	VenusProtocol/isolated-pools	78c5a09	contracts/Shortfall/IShortfall.sol	25cc4b1b91a3fdd1c93b04e58a4adf6fda043e6168bec1c8bfa6c384391c54bb
MPS	VenusProtocol/isolated-pools	78c5a09	contracts/test/Mocks/MockPancakeSwap.sol	9d6935d3fd193ccaaeaa161a5fd17fa7b9dd42694672042e56dbf0dae23a2aba
MPO	VenusProtocol/isolated-pools	78c5a09	contracts/test/Mocks/MockPriceOracle.sol	5f027734f88b622930f892d2d36632d86e5bc3c91daa466b103d9dbdf542924c
MTM	VenusProtocol/isolated-pools	78c5a09	contracts/test/Mocks/MockToken.sol	c537a3a35fd23354b5cd6fc253ef135b919cf61adab1301e55d5a2859a347349
CHV	VenusProtocol/isolated-pools	78c5a09	contracts/test/ComptrollerHarness.sol	393e185e4876c2a248014285d516a988dabe20b054d3279df14ffc731e442bb3
COP	VenusProtocol/isolated-pools	78c5a09	contracts/test/ComptrollerScenario.sol	2604039ebda4acc344e6a004ab96bb8524a6d00b31e2de61ee2dffbc73ae7abe
CON	VenusProtocol/isolated-pools	78c5a09	contracts/test/Const.sol	e2ac5c80b3c04222149761cd29a9c8e5b8ecbd09c56c35af10e8edc6deabb834
COU	VenusProtocol/isolated-pools	78c5a09	contracts/test/Counter.sol	8c2ffb5bea4945d4ae77f1a1dd5203d0212c6dcf2d19a950bf463e932188bd49
ERC	VenusProtocol/isolated-pools	78c5a09	contracts/test/ERC20.sol	e7aa577391e780c561c2e59d11591ba0fc3dbecce509204aacc08cb114b06e0f
ETV	VenusProtocol/isolated-pools	78c5a09	contracts/test/EvilToken.sol	d37d577751a6182edb5410fea272d2d5f9ca1a7ee5c4915f046aaa2d3ee46690
FMV	VenusProtocol/isolated-pools	78c5a09	contracts/test/FalseMarker.sol	8c2b8b0fb65b88c434cf807164ed2188c54376a96c721fd653d09c512931cb0f
FTV	VenusProtocol/isolated-pools	78c5a09	contracts/test/FaucetToken.sol	f178e51f0442ddf358ab138a2271e0cfdb902653f76dd8463ff1aa5b41b528dd

ID	Repo	Commit	File	SHA256 Checksum
FTP	VenusProtocol/isolated-pools	78c5a09	contracts/test/FeeToken.sol	4b4385e78ac88ac32d589f3ad1e2fc9d1b149fa69e5af7f51d622dc3df6bbdea
FPO	VenusProtocol/isolated-pools	78c5a09	contracts/test/FixedPriceOracle.sol	b7caa53eb755a4614799e4b66d04d05bb495da0b3aff603da028213d7c450ebd
HML	VenusProtocol/isolated-pools	78c5a09	contracts/test/HarnessMaxLoopsLimitHelper.sol	b54963cefcc7f017bdc3964053ec14b7cd0f4a49d5257157ed757c212f8f8896
IRH	VenusProtocol/isolated-pools	78c5a09	contracts/test/InterestRateModelHarness.sol	ee55287b58de7094c581375c736c04730cf897587fd7135eb6888207b89ccfc9
MHV	VenusProtocol/isolated-pools	78c5a09	contracts/test/MathHelpers.sol	fb5f9f90175b5d59a381ff10c280959061c2587dadc3e29a1c244609fb0ea57d
POP	VenusProtocol/isolated-pools	78c5a09	contracts/test/PriceOracleProxy.sol	20d23dd5e44a4db6f5190f40b53fb1a9aceb39f6554f1dea7d80032ed0dd60d2
SMV	VenusProtocol/isolated-pools	78c5a09	contracts/test/SafeMath.sol	89b0edc3a9d8af6906a7fb7149d8272dd1ae52e325aa8806df6df1bb2bd068c8
SPO	VenusProtocol/isolated-pools	78c5a09	contracts/test/SimplePriceOracle.sol	946ca73da4e4d38338f10bc694a2f78eb03bd49fed563f095195d1ae2ff24268
STR	VenusProtocol/isolated-pools	78c5a09	contracts/test/Structs.sol	f608a910f5987f282577960ad2a2263fd8bb9739a6f136e0bacc0f454efeb2bc
TIV	VenusProtocol/isolated-pools	78c5a09	contracts/test/TetherInterface.sol	410e91848c6a01bf7dc7a15e4fc63ac91278c257f5e4fbac892aeb83f0cafa6f
UVT	VenusProtocol/isolated-pools	78c5a09	contracts/test/UpgradedVToken.sol	9330cc3dc9efc86d956f3a597ff7a55db95ddda4ac084d2907993915661c23dc
VTH	VenusProtocol/isolated-pools	78c5a09	contracts/test/VTokenHarness.sol	3470a871f74ffe405e1e8cd74a02bcf7f8f46ccf1f334b05ab4d6c8fd6c58463
WBT	VenusProtocol/isolated-pools	78c5a09	contracts/test/WBTC.sol	8b00eeb310545d377d8903e979790f0ee3e53b7127d89afdf21900cb62cce04c

ID	Repo	Commit	File	SHA256 Checksum
BJM	VenusProtocol/isolated-pools	78c5a09	contracts/BaseJumpRateModelV2.sol	089da609505f30374eaa99b349d70f70055da454560566f9eaf5d0d4f48f5788
CIP	VenusProtocol/isolated-pools	78c5a09	contracts/ComptrollerInterface.sol	03544bfc5ffa24bd281953ee7636186578a8617b291d7b279de7d29ada49c0a5
ERP	VenusProtocol/isolated-pools	78c5a09	contracts/ErrorReporter.sol	293ce58a9fe2c4ee4c110e5c3f078bbe12e0da9cd4b156fc427e86fa34f3b2ff
ENV	VenusProtocol/isolated-pools	78c5a09	contracts/ExponentialNoError.sol	9d9f3eb5ffe5c5d12bb354be3d34f17f24d30368dde86bcda8c60d34d04496e6
IPR	VenusProtocol/isolated-pools	78c5a09	contracts/IPancakeswapV2Router.sol	c879c2c81af8a8139c685794b2be9945b421b72bf5bb14033c2d17a41ef7e34a
IRV	VenusProtocol/isolated-pools	78c5a09	contracts/InterestRateModel.sol	1003dbfdb6a8e3579db41a414d8a951f3aef397be84a9005ee181f41e1f8a76e
JRV	VenusProtocol/isolated-pools	78c5a09	contracts/JumpRateModelV2.sol	3b20ec2f216cb65b6981007ed42d542fefed9e430ce3292d65239e8714d062e0
VIV	VenusProtocol/isolated-pools	78c5a09	contracts/VTokenInterfaces.sol	b54c1bf95aa76d5c1cce507aa53e7f0a5c3267b0a0664eb163e9d5c18a8d857c

APPROACH & METHODS | VENUS - ISOLATED POOLS

This report has been prepared for Venus to discover issues and vulnerabilities in the source code of the Venus - Isolated Pools project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Manual Review and Static Analysis techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

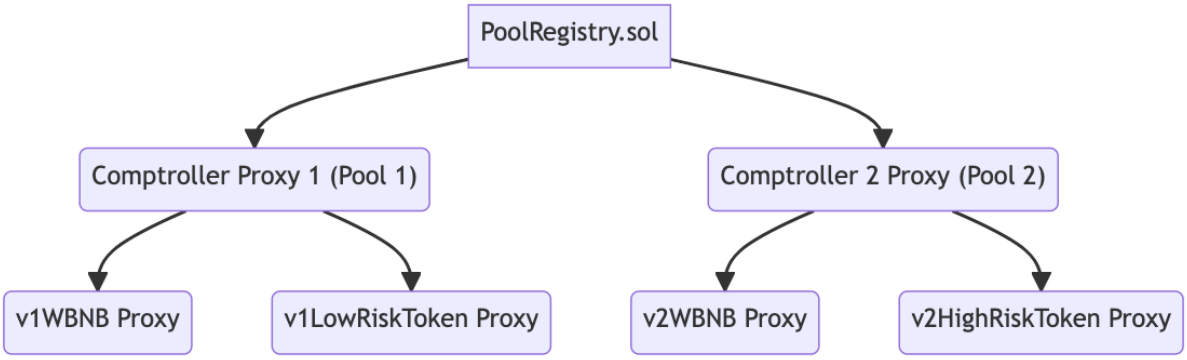
The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

REVIEW NOTES | VENUS - ISOLATED POOLS

System Overview

Venus - Isolated Pools is a lending protocol designed to distribute lending over multiple isolated pools to reduce the risks associated with having all available assets interact in one pool. Each pool maintains isolation by creating a unique `vToken` for each collateral asset and calculating its borrow limit independently of the collateral's use in other pools. For example, see the diagram below:



In the scenario, there are two pools created via the function `createRegistryPool()` in `PoolRegistry.sol`, and each pool contains two markets, added via the function `addMarket()` in `PoolRegistry.sol`. Each pool has a market for `WBNB`, where `v1WBNB` is the corresponding `vToken` for Pool 1, and `v2WBNB` is the corresponding `vToken` for Pool 2. In addition, Pool 1 has a market added for `LowRiskToken` and Pool 2 has a market added for `HighRiskToken`. Assume that the current supply rate for Pool 1 is less than the supply rate in Pool 2. In this case, a user who wants to supply `WBNB` can choose between supplying Pool 1 or Pool 2. If a user decides to supply Pool 1, then they will receive `v1WBNB` and gain a lower supply rate in exchange for taking on less risk. If a user supplies to Pool 2, then they will receive `v2WBNB` and gain a higher supply rate in exchange for taking on more risk. Since Pool 2 includes a higher risk token, this option involves exposure to more volatile market conditions. A participant should carefully research all tokens in a given pool as well as the risk associated with the each market's liquidation threshold and collateralization factor. In particular, the more volatile the token and the higher the collateralization factor and liquidation threshold, the more risk associated with the pool. Each pool is its own individual lending ecosystem, which means if you only supply assets to Pool 1, you can only borrow assets from Pool 1.

Contract Summaries

PoolRegistry

The `PoolRegistry` keeps track of the pools and the markets that have been added to each pool. This contract is how new pools are created, via `createRegistryPool()`, and how new markets are added to a pool, via `addMarket()`. The `owner` of this contract is the only entity allowed to create new pools or add new markets to pools.

Factories

There are three factory contracts:

- `JumpRateModelFactory`;
- `VTokenProxyFactory`;
- `WhiteRateInterestModelFactory`.

These contracts are designed to deploy contracts when adding markets to a pool. In particular, the `JumpRateModelFactory` and `WhiteRateInterestModelFactory` deploy contracts for the rate model based on which was chosen for the market. The `VtokenProxyFactory` is used to generate a new `vToken` proxy for each market when it is added to a pool.

Risk Fund

The risk fund concerns three main contracts:

- `ProtocolShareReserve`;
- `RiskFund`;
- `ReserveHelpers`.

The three contracts are designed to hold fees that have been accumulated from liquidations and spread, send a portion to the protocol treasury, and send the remainder to the `RiskFund`. When `reduceReserves()` is called in a `vToken` contract, all accumulated liquidation fees and spread are sent to the `ProtocolShareReserve` contract. Once funds are transferred to the `ProtocolShareReserve`, anyone can call `releaseFunds()` to transfer 70% to the `protocolIncome` address and the other 30% to the `RiskFund` contract. Once in the `RiskFund` contract, the tokens can be swapped via `PancakeSwap` pairs to the convertible base asset, which can be updated by the owner of the contract. When tokens are converted to the `convertibleBaseAsset`, they can be used in the `Shortfall` contract to auction off the pool's bad debt. Note that just as each pool is isolated, the risk funds for each pool are also isolated: only the risk fund for the same pool can be used when auctioning off the bad debt of the pool.

Shortfall

`Shortfall` is an auction contract designed to auction off the `convertibleBaseAsset` accumulated in `RiskFund`. The `convertibleBaseAsset` is auctioned in exchange for users paying off the pool's bad debt. An auction can be started by anyone once a pool's bad debt has reached a minimum value. This value is set and can be changed by the authorized accounts. If the pool's bad debt exceeds the risk fund plus a 10% incentive, then the auction winner is determined by who will

pay off the largest percentage of the pool's bad debt. The auction winner then exchanges for the entire risk fund. Otherwise, if the risk fund covers the pool's bad debt plus the 10% incentive, then the auction winner is determined by who will take the smallest percentage of the risk fund in exchange for paying off all the pool's bad debt.

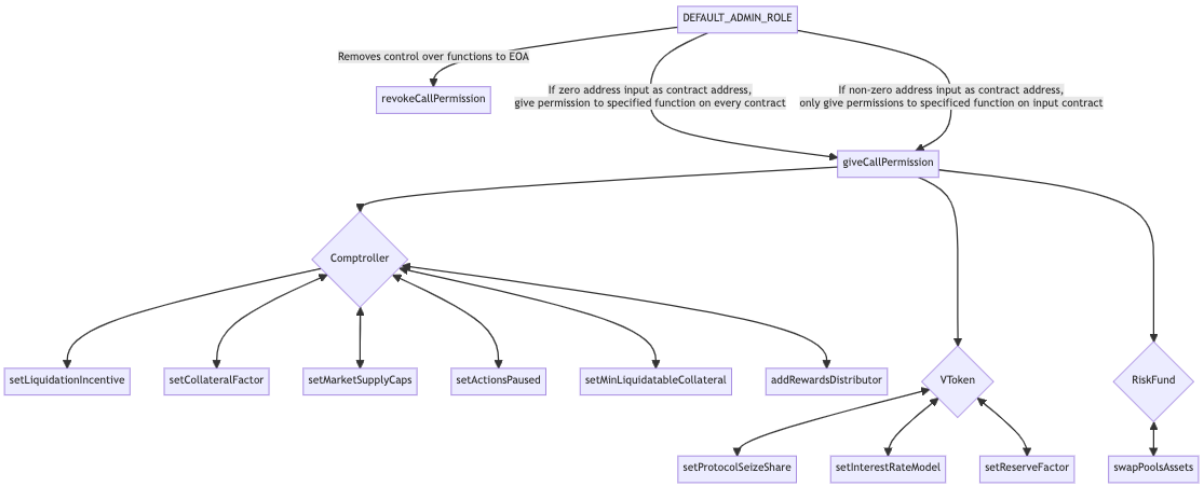
Rewards

Users can receive additional rewards through a `RewardsDistributor`. Each `RewardsDistributor` proxy is initialized with a specific reward token and `Comptroller`, which can then distribute the reward token to users that supply or borrow in the associated pool. Authorized users can set the reward token borrow and supply speeds for each market in the pool. This sets a fixed amount of reward token to be released each block for borrowers and suppliers, which is distributed based on a user's percentage of the borrows or supplies respectively. The owner can also set up reward distributions to contributor addresses (distinct from suppliers and borrowers) by setting their contributor reward token speed, which similarly allocates a fixed amount of reward token per block.

The owner has the ability to transfer any amount of reward tokens held by the contract to any other address. Rewards are not distributed automatically and must be claimed by a user calling `claimRewardToken()`. Users should be aware that it is up to the `owner` and other centralized entities to ensure that the `RewardsDistributor` holds enough tokens to distribute the accumulated rewards of users and contributors.

AccessControlManager

`AccessControlManager` inherits OpenZeppelin's AccessControl as a base. The contract gives specific addresses the ability to call certain functions in the contracts (see **Centralization Risks Finding** for more details). The `DEFAULT_ADMIN_ROLE` can grant or remove the ability of an address to call a function for a specific contract address or for all contracts that have the same function signature.



PoolLens

The `PoolLens` contract is designed to retrieve important information for each registered pool. A list of essential information for all pools within the lending protocol can be acquired through the function `getAllPools()`. Additionally, the following records can be looked up for specific pools and markets:

- the `vToken` balance of a given user;
- the pool data (oracle address, associated vToken, liquidation incentive, etc) of a pool via its associated comptroller address;
- the `vToken` address in a pool for a given asset;
- a list of all pools that support an asset;
- the underlying asset price of a `vToken`;
- the metadata (exchange/borrow/supply rate, total supply, collateral factor, etc) of any `vToken`.

Rate Models

These contracts help algorithmically determine the interest rate based on supply and demand. If the demand is low, then the interest rates should be lower. In times of high utilization, the interest rates should go up. As such, the lending market borrowers will earn interest equal to the borrowing rate multiplied by utilization ratio.

VToken

Each asset that is supported by a pool is integrated through an instance of the `VToken` contract. As outlined in the protocol overview, each isolated pool creates its own `vToken` corresponding to an asset. Within a given pool, each included `vToken` is referred to as a market of the pool. The main actions a user regularly interacts with in a market are:

- mint/redeem of `vToken`;
- transfer of `vToken`;
- borrow/repay a loan on an underlying asset;
- liquidate a borrow or liquidate/heal an account.

A user supplies the underlying asset to a pool by minting `vToken`, where the corresponding `vToken` amount is determined by the exchangeRate. The exchangeRate will change over time, dependent on a number of factors, some of which accrue interest. Additionally, once users have minted `vToken` in a pool, they can borrow any asset in the isolated pool by using their `vToken` as collateral. In order to borrow an asset or use a `vToken` as collateral, the user must be entered into each corresponding market (else, the `vToken` will not be considered collateral for a borrow). Note that a user may borrow up to a portion of their collateral determined by the market's collateral factor. However, if their borrowed amount exceeds an amount calculated using the market's corresponding liquidation threshold, the borrow is eligible for liquidation. When a user repays a borrow, they must also pay off interest accrued on the borrow.

The Venus protocol includes unique mechanisms for healing an account and liquidating an account. These actions are performed in the `Comptroller` and consider all borrows and collateral for which a given account is entered within a market. These functions may only be called on an account with a total collateral amount that is no larger than a universal

`minLiquidatableCollateral` value, which is used for all markets within a `Comptroller`. Both functions settle all of an account's borrows, but `healAccount()` may add `badDebt` to a `vToken`. For more detail, see the description of `healAccount()` and `liquidateAccount()` in the `Comptroller` summary section below.

Comptroller

The `Comptroller` is designed to provide checks for all minting, redeeming, transferring, borrowing, lending, repaying, liquidating, and seizing done by the `vToken` contract. Each pool has one `Comptroller` checking these interactions across markets. When a user interacts with a given market by one of these main actions, a call is made to a corresponding hook in the associated `Comptroller`, which either allows or reverts the transaction. These hooks also update supply and borrow rewards as they are called. The comptroller holds the logic for assessing liquidity snapshots of an account via the collateral factor and liquidation threshold. This check determines the collateral needed for a borrow, as well as how much of a borrow may be liquidated. A user may borrow a portion of their collateral with the maximum amount determined by the markets collateral factor. However, if their borrowed amount exceeds an amount calculated using the market's corresponding liquidation threshold, the borrow is eligible for liquidation.

The `Comptroller` also includes two functions `liquidateAccount()` and `healAccount()`, which are meant to handle accounts that do not exceed the `minLiquidatableCollateral` for the `Comptroller`:

- **healAccount()**: This function is called to seize all of a given user's collateral, requiring the `msg.sender` repay a certain percentage of the debt calculated by `collateral/(borrows*liquidationIncentive)`. The function can only be called if the calculated percentage does not exceed 100%, because otherwise no `badDebt` would be created and `liquidateAccount()` should be used instead. The difference in the actual amount of debt and debt paid off is recorded as `badDebt` for each market, which can then be auctioned off for the risk reserves of the associated pool.
- **liquidateAccount()**: This function can only be called if the collateral seized will cover all borrows of an account, as well as the liquidation incentive. Otherwise, the pool will incur bad debt, in which case the function `healAccount()` should be used instead. This function skips the logic verifying that the repay amount does not exceed the close factor.

DEPENDENCIES | VENUS - ISOLATED POOLS

Third Party Dependencies

The protocol is serving as the underlying entity to interact with third party protocols. The third parties that the contracts interact with are:

- PancakeSwap
- ERC20 Tokens

Recommendation

We understand that the business logic requires interaction with third parties. The scope of the audit treats third party entities as black boxes and assumes their functional correctness. However, in the real world, third parties can be compromised and this may lead to lost or stolen assets. In addition, upgrades of third parties can possibly create severe impacts, such as increasing fees of third parties, migrating to new LP pools, etc.

Furthermore, it is vital that the third party tokens are chosen with extreme caution. For example, there are choices for `underlying` that are known to be incompatible with this protocol and should not be allowed:

- `VToken` instances;
- LP tokens;
- Rebase Tokens.

We encourage the team to constantly monitor the statuses of any third parties the protocol interacts with to mitigate side effects when unexpected activities are observed.

Alleviation

`[CertiK]`: The client acknowledged the issue and stated the following.

`[Venus]`: "We'll define and follow guidelines in the election of the underlying tokens to discard noncompatible tokens."

Out Of Scope Dependencies

The `Comptroller` and `RiskFund` mechanics depend on the set `oracle` to function correctly. This contract is out-of-scope for this audit.

The scope of the audit treats out-of-scope dependencies as black boxes and assumes their functional correctness.

Recommendations

We recommend carefully auditing this contract and monitoring it to ensure its compatibility with the protocol.

Alleviation

`[CertiK]` : The client acknowledged the finding stating they have already audited the `oracle` contract.

FINDINGS | VENUS - ISOLATED POOLS

36	1	9	4	8	14
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for Venus - Isolated Pools. Through this audit, we have uncovered 36 issues ranging from different severity levels. Utilizing the techniques of Manual Review & Static Analysis to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
CVP-01	Improper Checks For `collateralFactor` And `liquidationThreshold`	Inconsistency	Critical	Resolved
COM-01	Use Of `_ensureMaxLoops()` May Lock Users' Positions	Logical Issue	Major	Resolved
CVP-02	Lack Of Validation On `LiquidationOrder[]`	Logical Issue	Major	Resolved
CVP-04	Possible Insufficient Checks In `liquidateAccount()` And `healAccount()`	Logical Issue	Major	Resolved
GLOBAL-01	**Centralization Related Risks**	**Centralization**	**Major**	**Mitigated**
GLOBAL-02	**Centralized Control Of Contract Upgrade**	**Centralization**	**Major**	**Mitigated**
RFF-02	`_swapAsset()` Returns Wrong Value If Path Has Length Greater Than 2	Logical Issue	Major	Resolved
VPB-01	Function `transferReserveForAuction()` May Transfer Reserves To Wrong Contract	Logical Issue	Major	Resolved
VPB-23	Potential Rounding Error Exploit	Logical Issue	Major	Resolved
VTV-01	Function `reduceReserves()` Is Unprotected	Logical Issue	Major	Resolved
COP-01	Case When Collateral Equals Scaled Borrows Is No Longer Covered	Logical Issue	Medium	Resolved

ID	Title	Category	Severity	Status
VPB-04	Interfaces Not Inherited And Improper/Missing Implementation	Logical Issue	Medium	Resolved
VPB-05	Lack Of Storage Gap In Upgradeable Contract	Logical Issue	Medium	Resolved
VTV-02	`mintBehalf()` Can Mint Tokens To The Zero Address	Logical Issue	Medium	Resolved
COP-05	Price Is Not Updated For All Users Markets In `preBorrowHook()`	Logical Issue	Minor	Resolved
PLV-01	Incorrect Calculation In `PoolLens`	Logical Issue	Minor	Resolved
RFR-01	Ineffective Check For `minAmountToConvert`	Logical Issue	Minor	Acknowledged
VPB-06	Missing Zero Address Validation	Logical Issue	Minor	Resolved
VPB-08	Changing `convertibleBaseAsset` Can Cause Issues	Logical Issue	Minor	Resolved
VPB-10	`initialize()` Is Unprotected And Inconsistent Init Function	control-flow	Minor	Resolved
VPH-01	Missing Input Validation	Volatile Code	Minor	Resolved
VTV-03	Approval Race Condition	Logical Issue	Minor	Resolved
COP-02	Possible Gas Issue With `updatePrices()`	Logical Issue	Informational	Acknowledged
CVP-05	Storage Variables Declared Outside Of `ComptrollerV1Storage`	Coding Style	Informational	Resolved
JRM-01	Unnecessary Inheritance	Logical Issue	Informational	Resolved
RDR-01	Anyone Can Claim Rewards For Another User	Logical Issue	Informational	Acknowledged

ID	Title	Category	Severity	Status
RFR-02	Duplicate Imports	Coding Style	Informational	Resolved
SSP-01	Inconsistent Comment And Code	Inconsistency	Informational	Resolved
VPB-02	Incompatibility With Fee On Transfer Tokens	Logical Issue	Informational	Acknowledged
VPB-11	Volatile Conditions	Logical Issue	Informational	Resolved
VPB-12	Missing Or Unused Emit Events	Coding Style	Informational	Resolved
VPB-13	Missing Access Restriction	control-flow	Informational	Resolved
VPB-19	Atypical Constructor Implementation	Coding Style	Informational	Resolved
VPB-21	Access Control Changes	Inconsistency	Informational	Resolved
VPH-02	Typos And Inconsistencies	Inconsistency	Informational	Resolved
VTV-04	Unnecessary Reference To `VTokenInterface`	Coding Style	Informational	Resolved

CVP-01 IMPROPER CHECKS FOR `collateralFactor` AND `liquidationThreshold`

Category	Severity	Location	Status
Inconsistency	Critical	contracts/Comptroller.sol (Base): 47~48, 754~762	Resolved

Description

The `collateralFactor` determines the maximum percentage a user can borrow against their collateral, while the `liquidationThreshold` is the maximum percentage of the collateral that the borrowed value can reach before the borrow is eligible for liquidation. This means that the `collateralFactor` should always be less than the `liquidationThreshold`, otherwise a user can borrow an amount and be eligible for liquidation at the same time (see scenario below). However, if `setCollateralFactor()` is called with such values it will revert due to the check:

```
760  if (newLiquidationThresholdMantissa > newCollateralFactorMantissa) {
761              revert InvalidLiquidationThreshold();
762          }
```

Scenario

Assume for a given `vToken`, say vUSDC, with `underlying` USDC as the asset, listed as a market, that the corresponding `collateralFactorMantissa` is 0.9 (the maximum) while the corresponding `liquidationThresholdMantissa` is 0.85. Suppose the exchange rate is 1 USDC to 10 vUSDC. Further, assume that the `minLiquidatableCollateral` is 95 (USD) for simplicity.

1. User Alice supplies 100 USDC and receives 1000 vUSDC in return.
2. Alice wants to borrow against her vUSDC (this is the only collateral she has and this is her first borrow). She attempts to borrow 90 USDC, worth 90 USD. which is equivalent to 90% of the worth of her vUSDC. Since this is equivalent to the worth of the `snapshot.weightedCollateral`, the `shortfall` of the snapshot is 0, and the borrow is successful.
3. User Bob attempts to liquidate Alice through `liquidateBorrow()`. Since Alice's `totalCollateral` is worth 100 USD, this is the appropriate function to call. Through this call, the `liquidationThresholdMantissa` is used as the weight, and now `snapshot.weightedCollateral` is calculated to be 85. However, the `borrowPlusEffects` sum will still be equal to 90. Since `borrowsPlusEffects` is greater than the `weightedCollateral`, the `shortfall` value is the positive difference of 5, allowing the liquidation to successfully complete.

As a conclusion, a portion of Alice's borrow that she was allowed to make can immediately be liquidated.

Recommendation

We recommend requiring the `liquidationThreshold` is greater than the `collateralFactor`. In addition, we recommend implementing a maximum value for the `liquidationThreshold` that takes the liquidation incentive into account and that gives liquidators time to liquidate borrows before the protocol incurs bad debt.

Alleviation

`[CertiK]`: The client resolved the issue by changing the inequality and reverting if the `newLiquidationThresholdMantissa` is greater than 1 in commits: 934e7fd050814280cb8ac603e16447d10656d087 and 7cc7b68b044f9446d0e4223274c2d9ad4cbc67c3.

COM-01 USE OF `_ensureMaxLoops()` MAY LOCK USERS' POSITIONS

Category	Severity	Location	Status
Logical Issue	Major	contracts/Comptroller.sol (Addendum Base): 157~158, 214~215, 274~275, 306~307, 363~364, 380~381, 408~409, 529~530, 568~569, 594~595, 704~705, 835~836, 950, 965, 1207~1208, 1220~1221, 1267~1268, 1319~1320	Resolved

Description

Use of `_ensureMaxLoops()` on `accountAssets` Array

Function `enterMarkets()` makes a check that length `vTokens.length` does not exceed a maximum limit through function `_ensureMaxLoops()`. The `enterMarkets()` function does not make any checks on whether the total number of markets entered for the user, ie, the length of `accountAssets`, exceeds the limit in `_ensureMaxLoops()`.

This discrepancy allows the limit to be passed in `enterMarkets()`, but any other function relying on the check will revert. This means that `exitMarket()` cannot be called to reduce the number of markets the user is in, and, in particular, the function `_getHypotheticalLiquiditySnapshot()` reverts because of the `_ensureMaxLoops()` check, preventing a user from redeeming or borrowing. As a result, their position is trapped in the protocol.

We note this issue is less likely to occur, unless the length of `allMarkets` has already exceeded the `maxLoopsLimit`, too, which is possible (see the recommendation below).

Use of `_ensureMaxLoops()` on `RewardsDistributors` Array

Function `addRewardsDistributor()` checks that the `rewardsDistributorsLength` does not exceed a given limit through use of `_ensureMaxLoops()`, but this value is incremented after the check. This means that the `rewardsDistributorsLength` may not exceed the `maxLoopsLimit`, but the updated value after the increment may. As a result, Any function checking that `_ensureMaxLoops()` is successful on the length of the `RewardsDistributors` array will cause a revert. Since the array cannot be decremented, any surpassing of the limit would cause the corresponding functions to stop working.

Proof of Concept

1. Assume the `maxLoopsLimit` of the `Comptroller` is 10 for simplicity. Further, assume that the length of array `allMarkets` is 11, so that there are eleven markets available for entering.
2. Alice calls `enterMarkets()` with an address array `vTokens` of length 10.
3. The check to `_ensureMaxLoops(len)` where `len = vTokens.length` passes, since the length of the array does not exceed 10. Alice is now entered into 10 of the 11 total markets.

4. Alice calls `enterMarkets()` again with the last remaining `vToken` for the address array. Now, the length of the input array is 1, again passing the `_ensureMaxLoops()` check.
5. The array `accountAssets[borrower]` where Alice is the borrower is now a length of 11.
6. As a result, if Alice attempts to call `redeem()` or `borrow()` at this stage, she will be unable to, since these functions interact with functions `_getHypotheticalLiquiditySnapshot()`, where `_ensureMaxLoops()` will revert due to the length of Alice's `accountAssets` array exceeding a length of 10. Alice's position cannot be liquidated or transferred either, for the same reason.

Recommendation

Recommendation for `accountAssets` Array

We recommend adding a check in `enterMarkets()` that the total length of `accountAssets[msg.sender]` does not exceed the limit enforced by `ensureMaxLoops()`. The current check on input `vTokens.length` can be kept in `enterMarkets()`. With this addition, the check of `_ensureMaxLoops()` on `accountAssets` can be removed from the following functions:

- `exitMarket()`;
- `healAccount()`;
- `liquidateAccount()` (the check on `orders.length` can remain in the source code);
- `_getHypotheticalLiquiditySnapshot()`;

Recommendation for `RewardsDistributors` Array

We recommend replacing the check of `_ensureMaxLoops()` in `addRewardsDistributor()` with a check through `_ensureMaxLoops()` for the updated length of `rewardsDistributors` after the new `_rewardsDistributor` has been added to the array. This ensures that the limit is not exceeded through the addition. With this addition, the check of `_ensureMaxLoops()` on `rewardsDistributors` can be removed from the following functions:

- `preMintHook()`;
- `preRedeemHook()`;
- `preBorrowHook()`;
- `preRepayHook()`;
- `preSeizeHook()`;
- `preTransferHook()`;
- `supportMarket()`;

Additionally, the check that `marketsCount` does not exceed the `maxLoopLimit` should be replaced with a check in `_addMarket()` that the total length of `allMarkets` does not exceed the `maxLoopLimit` after `vToken` is pushed to the array.

Alleviation

`[CertiK]` : The client made the recommended changes in commits 2cb780e525c1c6c836db1d416c2328fb2c560734 and da284b4a01828ea1ae4c13b207ba7d2e73ceec0b.

CVP-02 | LACK OF VALIDATION ON `LiquidationOrder[]`

Category	Severity	Location	Status
Logical Issue	Major	contracts/Comptroller.sol (Base): 675	Resolved

Description

The function `liquidateAccount()` allows the `msg.sender` to liquidate the input `borrower` in multiple markets by inputting a dynamic `LiquidationOrder` array, `orders`. However, the input `LiquidationOrder` components, `vTokenCollateral` and `vTokenBorrowed`, are not checked for each entry of the array, and external calls are made to them, allowing an attack contract to be called.

Scenario

A malicious user can create an attack contract with the same function names as those in a legitimate `vToken` contract, but with logic used to interact with the Venus protocol in unexpected ways. If the contract is updated with new functionality, then this entry point for a malicious user could cause significant harm to the protocol.

Recommendation

We recommend checking each input `LiquidationOrder` of the `orders` array to verify that each component `vTokenCollateral` and `vTokenBorrowed` are listed markets.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 55a1eac4505ed3170dafdd4801e13a2c6bd7ab87.

CVP-04 POSSIBLE INSUFFICIENT CHECKS IN `liquidateAccount()` AND `healAccount()`

Category	Severity	Location	Status
Logical Issue	Major	contracts/Comptroller.sol (Base): 689~690	Resolved

Description

Function `liquidateAccount()`

When function `liquidateAccount()` is called, the only liquidity snapshot check that is performed is whether `collateralToSeize` is at least as large as `snapshot.totalCollateral`. In such a case, the function reverts. Since the snapshot was already assessed, when `forceLiquidateBorrow()` is called on each `vTokenBorrowed` address, the fact that `skipLiquidityCheck` is set to `true` means that further assessment will not be performed on the snapshot. In particular, the check that `shortfall` is nonzero is not performed, meaning those that have a `shortfall` of zero can be liquidated via `liquidateAccount()` when their `totalCollateral` is less or equal to `minLiquidatableCollateral`.

The `collateralToSeize` value is the `snapshot.borrows` value multiplied by the `liquidationIncentiveMantissa`. In the case where `liquidationIncentiveMantissa` is larger than `1e18`, the `collateralToSeize` will be larger than the value of `snapshot.borrows`; the fact that `collateralToSeize` is larger or equal to `snapshot.totalCollateral` does not imply that `snapshot.totalBorrows` is greater or equal to `snapshot.totalCollateral` in that case.

With this set up, it is possible that the `borrows` of a `borrower` account do not exceed the `totalCollateral` and that `shortfall` is 0, but that this check still passes, allowing the `borrower` account to be liquidated. In such a case, the function `liquidateAccount()` is successfully called on a borrower that should not have been liquidated. It appears this can be called on any account in which `totalCollateral` does not exceed the `minLiquidatableCollateral`.

Function `healAccount()`

Function `healAccount()` can be called for an account in which `totalCollateral` does not exceed the `minLiquidatableCollateral`, and when the account's `borrows` multiplied by the `liquidationIncentiveMantissa` is at least the `totalCollateral`. Again, these are the only checks performed on the snapshot for the account within this function. It is still possible that this function can be called for accounts in which the `shortfall` value is 0, since the check in `healAccount()` scales the account's borrows, and the check in `_getHypotheticalLiquiditySnapshot()` scales the account's `totalCollateral` value to determine the corresponding `shortfall`.

Recommendation

We recommend adding checks to ensure that only accounts with a shortfall are eligible to have `liquidateAccount()` or `healAccount()` called on them.

Alleviation

`[CertiK]` : The client made the recommended changes in commit: 1c8cc30e72686c525b2f68ec5e2b493f1405823c.

GLOBAL-01 | CENTRALIZATION RELATED RISKS

Category	Severity	Location	Status
Centralization	**Major**		**Mitigated**

Description

Shortfall.sol

In the contract `Shortfall`, the role `onlyOwner()` has the authority over the function `updatePoolRegistry()`. Any compromise to the `onlyOwner` account may allow the hacker to take advantage of this authority and update the `poolRegistry` contract address.

In addition, the role `DEFAULT_ADMIN_ROLE` can grant addresses the privilege to call the following functions in the contract `Shortfall`:

- `updateNextBidderBlockLimit()`
- `updateIncentiveBps()`
- `updateMinimumPoolBadDebt()`
- `updateWaitForFirstBidder()`

Any compromise to the `DEFAULT_ADMIN_ROLE` or these privileged functions may allow the hacker to take advantage of this authority and do the following:

- Update the time a bidder has to make the next bid before an auction can be closed;
- Update the incentive BPS;
- Update the minimum pool debt needed to start the auction;
- Update the time needed for the first bid before the auction will need to be restarted;

BaseJumpRateModelV2.sol

The role `DEFAULT_ADMIN_ROLE` can grant addresses the privilege to call the following functions in the contract `BaseJumpRateModelV2`:

- `updateJumpRateModel()`

Any compromise to the `DEFAULT_ADMIN_ROLE` or these privileged functions may allow the hacker to take advantage of this authority and change the `baseRatePerYear`, `multiplierPerYear`, `jumpMultiplierPerYear`, and `kink`. This may allow the hacker to change rates to collect more interest on supply, pay less interest on borrows, or sabotage the protocol.

PoolRegistry.sol

The role `DEFAULT_ADMIN_ROLE` can grant addresses the privilege to call the following functions in the contract `PoolRegistry`:

- `addMarket()`
- `createRegistryPool()`
- `updatePoolMetadata()`
- `setPoolName()`

Any compromise to the `DEFAULT_ADMIN_ROLE` or these privileged functions may allow the hacker to take advantage of this authority and do the following:

- Add a new market with rates and parameters they set. A hacker could use this to add a market for a coin they control and manipulate the price to take suppliers collateral and cause the protocol to incur large amounts of bad debt;
- Create a new registry pool with parameters they set including the beacon address. A hacker could take advantage of this by inputting a beacon address that points to a malicious comptroller implementation allowing them to take any funds supplied to the pool;
- Update a pools metadata to change the risk rating of the pool, category, logo URL, and description;
- Change the name of a pool;

RewardsDistributor.sol

The role `DEFAULT_ADMIN_ROLE` can grant addresses the privilege to call the function `setRewardTokenSpeeds()` in the contract `RewardsDistributor`. Any compromise to the `DEFAULT_ADMIN_ROLE` or these privileged functions may allow the hacker to take advantage of this authority and change the reward token speed to any value.

In the contract `RewardsDistributor` the role `onlyOwner` has authority over the following functions:

- `setContributorRewardTokenSpeed()`
- `setMaxLoopsLimit()`
- `grantRewardToken()`

Any compromise to the `onlyOwner` account may allow the hacker to take advantage of this authority and do the following:

- Change the contributor reward token speed to any value;
- Change the max loops, which limits that amount of `vToken` that `claimRewardToken()` can be called on at one time;
- Grant any amount of reward tokens, provided enough are held by the contract, to any user.

In addition, the `onlyOwner` is responsible for ensuring the contract has enough of the reward token to allow users to claim the rewards they have earned. Either by ensuring enough tokens are supplied to the contract or that the functions above are used in a way that will not exceed the contracts reward token balance.

ProtocolShareReserve.sol

In the contract `ProtocolShareReserve` the role `onlyOwner` has authority over the function `setPoolRegistry()`. Any compromise to the `onlyOwner` account may allow the hacker to take advantage of this authority and change the `poolRegistry` address.

RiskFund.sol

In the contract `RiskFund` the role `onlyOwner` has authority over the following functions:

- `setPoolRegistry()`
- `setShortfallContractAddress()`
- `setPancakeSwapRouter()`
- `setMaxLoopsLimit()`

Any compromise to the `onlyOwner` account may allow the hacker to take advantage of this authority and do the following:

- Set the `poolRegistry` address to a contract they control
- Set the `shortfall` contract address to an address they control to steal any reserve that is supposed to be transferred for auction;
- Set the PancakeSwap router to a malicious contract to steal any tokens that are intended to be swapped;
- Change the max loops, which limits the amount of `markets` that can be input into `swapPoolsAssets()`.

In the contract `RiskFund` the role `shortfall` has authority over the function `transferReserveForAuction()`. Any compromise to the `shortfall` account may allow the hacker to take advantage of this authority and transfer reserves to themselves.

The role `DEFAULT_ADMIN_ROLE` can grant addresses the privilege to call the following functions in the contract `RiskFund`.

- `swapPoolsAssets()`
- `setMinAmountToConvert()`

Any compromise to the `DEFAULT_ADMIN_ROLE` or these privileged functions may allow the hacker to take advantage of this authority and do the following:

- Swap a pools assets to the convertible base asset as well as set the minimum output amounts. If a hacker gained access to this, they can set a low minimum output amount and manipulate the pool to steal assets;
- Set the minimum amount that must be converted in order to swap assets.

Comptroller.sol

The role `DEFAULT_ADMIN_ROLE` can grant addresses the privilege to call the following functions in the contract `Comptroller`:

- `setCloseFactor()`
- `setCollateralFactor()`
- `setLiquidationIncentive()`
- `setMarketBorrowCaps()`
- `setMarketSupplyCaps()`
- `setActionsPaused()`
- `setMinLiquidatableCollateral()`

Any compromised to the `DEFAULT_ADMIN_ROLE` could allow the hacker to grant or revoke the privilege to call the aforementioned function, this may allow the hacker to take advantage of this authority and do the following:

- Change the value of `collateralFactorMantissa` and `liquidationThresholdMantissa` for any market, allowing market manipulation.
- Set any value as the new `liquidationIncentiveMantissa`.
- Change the maximum amount that can be borrowed for a specific underlying token.
- Change the amount of a specific VToken that can be minted.
- Pause any market from doing any action.
- Set any value as the `minLiquidatableCollateral`.
- Set any value as the `closeFactorMantissa`, especially lower than `closeFactorMinMantissa` or higher than `closeFactorMaxMantissa` to sabotage the contract.

These functions could be used to potentially attack the contract through market manipulation or to sabotage the protocol.

In the contract `Comptroller` the role `owner` has authority over the following functions:

- `addRewardsDistributor()`
- `setPriceOracle()`
- `setMaxLoopsLimit()`

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and do the following:

- Add a malicious contract as a `RewardsDistributor`.
- Set a malicious contract as the `oracle`.
- Set the `maxLoopsLimit` to 0, preventing use of the protocol.

VToken.sol

The role `DEFAULT_ADMIN_ROLE` can grant addresses the privilege to call the following functions in the contract `VToken`:

- `setProtocolSeizeShare()`
- `setReserveFactor()`

- `setInterestRateModel()`

Any compromised to the `DEFAULT_ADMIN_ROLE` could allow the hacker to grant or revoke the privilege to call the aforementioned function, this may allow the hacker to take advantage of this authority and do the following:

- Change the value of `protocolSeizeShareMantissa`, if the value is too high this could remove the incentives to perform liquidation.
- Change the value of `reserveFactorMantissa` to impact the growth of the reserves when accruing interest.
- Set a malicious contract as the `interestRateModel` to manipulate the borrow rate when accruing interest.

These functions could be used to potentially attack the contract through market manipulation or sabotage the protocol.

The role `owner` has authority over the function:

- `sweepToken()`

Any compromise to the `owner` account may allow the hacker to take advantage of this authority and do the following:

- Transfer non-underlying tokens in the contract to any address the hacker controls.

AccessControlled.sol

In the contract `AccessContolled` the role `owner` has authority over the following functions:

- `setAccessControlManager()`

Any compromise to the `owner` role may allow the hacker to take advantage and update the protocol to an `AccessControlManager` contract which allows them to completely control all functionality of the protocol.

AccessControlManager.sol

In the contract `AccessControlManager` the role `DEFAULT_ADMIN_ROLE` has authority over the following functions:

- `giveCallPermission()`
- `revokeCallPermission()`

Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow the hacker to take advantage and give an owned account complete permission of all relevant privileged functions within the protocol.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We recommend carefully managing the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend

centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement;
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles;
  OR
- Remove the risky functionality.

Alleviation

`[CertiK]` : The client has mitigated this finding with the use of Timelock and Dao. Please see the information below for the relevant information.

`[Venus]` : We have two main access control mechanisms:

(A) Ownership: using the onlyOwner modifier

(B) ACM: delegating the access control in the AccessControlManager contract deployed at 0x4788629abc6cfca10f9f969efdeaa1cf70c23555

The main criterion we are following to use (A) or (B) is:

- if the function is used to configure a protocol parameter (i.e. setReserveFactor()), we should use the ACM contract to allow us to launch a Normal/Fast-track/Critical VIP
- if the function is used to configure another contract (i.e. Comptroller.addRewardsDistributor()), we should use the Ownership mechanism.

Specifically, we have modified our codebase, to integrate ACM in the following functions:

Shortfall:

- startAuction
- restartAuction

BaseJumpRateModelV2:

- updateJumpRateModel

PoolRegistry:

- addMarket
- createRegistryPool
- updatePoolMetadata
- setPoolName

RewardsDistributor:

- setRewardTokenSpeeds

RiskFund:

- setMinAmountToConvert

Comptroller:

- setCloseFactor

These changes have been made in this PR: https://github.com/VenusProtocol/isolated-pools/pull/180.

The Ownership of every listed contract will be transferred to the Normal Timelock contract, used to trigger VIP (Venus Improvement Proposals). This smart contract is already deployed at 0x939bD8d64c0A9583A7Dcea9933f7b21697ab6396.

And it's already used for the current VIP: https://app.venus.io/governance/.

The addresses for the FastTrack and Critical Timelock contracts are:

- Fast-track: 0x555ba73dB1b006F3f2C7dB7126d6e4343aDBce02
- Critical: 0x213c446ec11e45b15a6E29C1C1b402B8897f606d

The configurations for each type of VIP are as follows:

- normal: 24 hours voting + 48 hours delay
- fast-track: 24 hours voting + 6 hours delay
- critical: 6 hours voting + 1 hour delay

The initial set of functions that Fast-track and Critical will be given access to are:

- Comptroller.setCollateralFactor
- Comptroller.setMarketBorrowCaps
- Comptroller.setMarketSupplyCaps
- Comptroller.setActionsPaused

GLOBAL-02 | CENTRALIZED CONTROL OF CONTRACT UPGRADE

Category	Severity	Location	Status
Centralization	**Major**		**Mitigated**

Description

The Venus Isolated Pools protocol contains upgradeable contracts, the owner can upgrade the contracts at any time without the community's commitment. If an attacker compromises the account, they can change the implementation of the contract and drain tokens from the contract.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND

- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

`[Venus]` : The ownership of these contracts will be transferred to 0x939bd8d64c0a9583a7dcea9933f7b21697ab6396, that is the Timelock contract used to execute the normal Venus Improvement Proposals (VIP).

For normal VIPs, the time config is: 24 hours voting + 48 hours delay before the execution.

So, these contracts will be upgraded only via a Normal VIP, involving the community in the process.

RFF-02 `_swapAsset()` RETURNS WRONG VALUE IF PATH HAS LENGTH GREATER THAN 2

Category	Severity	Location	Status
Logical Issue	Major	contracts/RiskFund/RiskFund.sol (Addendum Base): 276	Resolved

Description

The function `_swapAsset()` uses the input `path` when swapping with the `pancakeSwapRouter`. However, if the `underlyingAsset` is not the `convertibleBaseAsset` the total amount is always set to be `amounts[1]`. However, this is not the amount of the `convertibleBaseAsset` that will be received if the input path has a length greater than 2. This will cause the wrong amount to be added to the `poolReserves` causing the value to be larger or smaller than expected. If the value is smaller than expected, then the contract will lock some of the `convertibleBaseAsset` in the contract as the balance stored will be less than the actual amount held by the contract.

Proof of Concept

Assume that the input path is `[underlyingAsset, intermediateAsset, convertibleBaseAsset]`.

- `amounts[0]` will be the input amount of `underlyingAsset`.
- `amounts[1]` will be the amount of `intermediateAsset` from first swapping the input amount of `underlyingAsset` to `intermediateAsset`.
- `amounts[2]` will be the amount of `convertibleBaseAsset` received from swapping the `amount[1]` of `intermediateAsset` to `convertibleBaseAsset`.

This demonstrates how the amount of `convertibleBaseAsset` received will not allways be `amounts[1]`.

Recommendation

We recommend using `amounts[path.length - 1]` as this is the amount of `convertibleBaseAsset` that will be received after the swap for any input path.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 70f2823e0024d237bd8d1d50910af924aa55f068.

VPB-01 FUNCTION `transferReserveForAuction()` MAY TRANSFER RESERVES TO WRONG CONTRACT

Category	Severity	Location	Status
Logical Issue	Major	contracts/RiskFund/RiskFund.sol (Base): 110, 180; contracts/Shortfall/Shortfall.sol (Base): 229, 233	Resolved

Description

The function `transferReserveForAuction()` checks if the `msg.sender` is the `shortfall` address. After the `shortfall` address calls this function, it will transfer the funds to `auctionContractAddress`. If `setAuctionContractAddress()` is called and these two addresses differ then the funds will be sent to the wrong address when `transferReserveForAuction()` is called in `Shortfall.sol`.

Recommendation

We recommend rechecking the logic and replacing the `auctionContractAddress` with the `shortfall` address. However, if this contract is intended to be used for multiple auction contracts, then a different check would be necessary to avoid errors.

Alleviation

`[CertiK]`: The client resolved this finding by merging the `shortfall` and `auctionContractAddress` in commit: 8a29a34b1bef57fd12f79ec431de9d0212de5581.

VPB-23 | POTENTIAL ROUNDING ERROR EXPLOIT

Category	Severity	Location	Status
Logical Issue	Major	contracts/Pool/PoolRegistry.sol (Addendum Base): 269~272; contracts/VToken.sol (Addendum Base): 1423~1442	Resolved

Description

If there is a single supplier left in any market, then they can cause the `totalSupply` to be very small and take advantage of a rounding error when the exchange rate is calculated. If the single supplier is malicious they can use this to drain the entire pools funds.

Scenario

Lets assume that a user manipulates the pools so that its total supply of the `vToken` is 2, with say 100_000 cash, 0 borrowed, 0 bad debt, and 0 in reserve.

- The exchange rate will be 100_000/2 = 50_000.
- The user then borrows 10_000 in another market in the same pool against it.
- The user then calls `redeemUnderlying(99_999)` which uses the following logic:

```
else {
            /*
             * We get the current exchange rate and calculate the amount to be
redeemed:
             *  redeemTokens = redeemAmountIn / exchangeRate
             *  redeemAmount = redeemAmountIn
             */
            redeemTokens = div_(redeemAmountIn, exchangeRate);
            redeemAmount = redeemAmountIn;
        }
```

So that redeemTokens = 99_999/50_000 = 1.99998 =1 as it will round down. Thus only 1 token will need to be redeemed for it. It will then check the redeem on the 1 token, allowing `_checkRedeemAllowed` to pass because the borrow is still over-collateralized according to the exchange rate.

Thus the user now has 99_999 tokens from redeeming plus the 10_000 they borrowed. So that they have profited 9_999.

This attack is only possible if the user is the only supplier so that they can manipulate the total supply, so that if there is an initial supply locked it is not feasible. Thus it should be ensured that the initial supply mints a sufficient amount so that rounding errors cannot be abused.

Recommendation

We recommend ensuring that the total supply of a `vToken` can never be below some minimum threshold, to prevent the exchange rate from being manipulated using this rounding error. One possible solution is to lock an amount of the `vToken` so that the total supply can never fall below the the locked balance.

Alleviation

`[CertiK]` : It is noted that in commit f075e8256a5215d438ff610f34cd3e25eea7c79d, the team has made adjustments in `PoolRegistry.sol` so that now a `vTokenReceiver` is specified each time a market is added. The `vTokenReceiver` address is minted an `amountToSupply` (based off the `initialSupply`) of vTokens through the `mintBehalf()` function in the `vToken` contract. Previously, the `msg.sender` was the address used in the `mintBehalf()` call.

`[Venus]` : The `vTokenReceiver` can be different each time. If the initial liquidity is provided by a 3rd party, this address will be an address provided by this third party. Just in case Venus would want to provide initial liquidity to some pool, it could use the Treasury contract (deployed at 0xf322942f644a996a617bd29c16bd7d231d9f35e9) as `vTokenReceiver` .

`[CertiK]` : The client resolved the attack scenario by rounding up in commit: aa966900ec5112e2e000ab25c2da851f3803489e. It should be noted, that because the amount of `vToken` to redeem is rounded up that if there is a low total supply a users funds are at risk as opposed to the protocols. However, it is unlikely that the total supply reaches such low levels unless deliberate steps are taken by the user. We do recommend users to carefully monitor the total supply and choose redeem amounts carefully if the supply is low to prevent a loss of funds when redeeming.

VTV-01 | FUNCTION `reduceReserves()` IS UNPROTECTED

Category	Severity	Location	Status
Logical Issue	Major	contracts/VToken.sol (Base): 449~450	Resolved

Description

The state variable `totalReserves` of the `VToken` contract becomes nonzero in one of three ways:

1. A borrower is liquidated - a portion of the corresponding seized amount of collateral defined by the `protocolSeizeShareMantissa` is sectioned for `totalReserves` (by default, this portion is 5% of the seized amount);
2. The function `addReserves()` is called - the caller supplies a corresponding amount of `underlying` token;
3. Through accumulated interest in function `accrueInterest()`

The `totalReserves` is a `uint` that represents the portion of the contract's underlying balance which may be taken out of the contract and sent to the corresponding `protocolShareReserves` contract.

Over time, `totalReserves` will accumulate interest with each call to the function `accrueInterest()`. The `borrowRateMantissa` is used in determining this rate of growth. It is necessary that the `totalReserves` value is kept relatively low, as accumulation over time may cause the `exchangeRate` to decrease towards 0. As such, it is assumed that in most cases the value of `totalReserves` is frequently reduced through function `reduceReserves()`.

While the `underlying` balance of the contract is updated to reflect the update to `totalReserves` when either `addReserves()` or `seize()` is called, the `underlying` balance does not accumulate interest as `totalReserves` accumulates interest in updates through `accrueInterest()`. If `totalReserves` is not periodically reduced, the `totalReserves` value can account for more of the `underlying` balance of the contract over time.

The fact that the function `reduceReserves()` is unprotected allows for the following attack vector.

Scenario

Assume that `totalReserves` is approximately equal to the contract's balance of the `underlying` asset (`totalReserves` may be less or equal to this value). This may occur in periods of high volumes of borrowing, or because `reduceReserves()` has not been called for a long period of time and interest has been allowed to accrue on `totalReserves`.

1. A user notices that `totalReserves` is approximately equal to the contract's balance of the `underlying` asset.
2. This user calls `reduceReserves()`, specifying a `reduceAmount` which is no more than the `totalReserves` value, and no more than the contract's balance of the `underlying`.
3. The transaction is successful, meaning most of the balance of the `underlying` asset is transferred to the corresponding `protocolShareReserves` contract.
4. As a result, users cannot currently borrow or redeem.

5. Within the `protocolShareReserves` contract, the `owner` can call `releaseFunds()`; this only grants access to 70% of the transferred value; the remaining amount is sent to `RiskFund` where it can only be auctioned.
6. Consequently, at most, the privileged accounts can add back in 70% of the underlying that was originally present. If they do this through `addReserves()`, then the above outlined sequence of events can be performed again; each time, less `underlying` is available to be added back to the `VToken` contract. Eventually, effectively all is sent to auction.

The `owner` of `protocolShareReserves` would have no choice but to either directly transfer the `underlying` to the `VToken` contract without using the `VToken` functionality, or they would have to `mint()` vTokens to add the `underlying` back. Neither option allows for restoration of the `totalReserves` value.

Recommendation

We recommend making the `reduceReserves()` function a privileged function, since it makes sensitive updates to the contract's state.

Alleviation

`[CertiK]`: The client made the following statement:

`[Venus]`: "We prefer to keep the reduceReserves function open (executable by anyone), to facilitate their integration with external agents/scripts that would automatise the income distribution."

`[CertiK]`: We recommend, if this function is not to be privileged, that systems are put in place to ensure that `reduceReserves()` is called frequently to avoid the `totalReserves` value from getting too large and causing issues.

`[CertiK]`: The team added checks that `borrowAmount < cash - reserves` and `redeemAmount < cash - reserves` respectively in functions `_borrowFresh()` and `_redeemFresh()`, in commit 78c5a0947dda78e3df87ad16b6f2a4521ee12968. While these checks mitigate some of the issues described above, the issue may still persist in the case where the `totalReserves` accrue interest over a large enough period of time so that the `totalReserves` reaches the value of `_getCashPrior()` without any user activity. In that case, the issue described above can be considered resolved with a plan outlined for regularly calling `reduceReserves()` to ensure that `totalReserves` does not grow large enough for an independent actor to engage with this function maliciously.

`[CertiK]`: The team states that they have a plan to automatically move the reserves from markets to other contracts where tokenomic rules will be applied. Transfers of reserves will be initiated both by incentivized agents and by the team at regular intervals. The team additionally states they will not only periodically execute `reduceReserves()` but will also plan on executing the function when risk is associated.

COP-01 CASE WHEN COLLATERAL EQUALS SCALED BORROWS IS NO LONGER COVERED

Category	Severity	Location	Status
Logical Issue	Medium	contracts/Comptroller.sol (update7): 628~630, 676~680	Resolved

Description

The function `healAccount()` is only callable when

```
1 > percentage
```

Where `percentage = div_(collateral, scaledBorrows)`. So that it covers the cases where `scaledBorrows > collateral`.

The function `liquidateAccount()` is only callable when

```
collateralToSeize < snapshot.totalCollateral
```

where `collateralToSeize` is the scaled borrows and `snapshot.totalCollateral` is the collateral. Thus it covers the cases where `scaledBorrows < collateral`.

As `healAccount()` and `liquidateAccount` must be the functions called to perform liquidations when `snapshot.totalCollateral <= minLiquidatableCollateral` if this is the case and `scaledBorrows = collateral` both of these functions will revert.

Thus the case when `snapshot.totalCollateral <= minLiquidatableCollateral` and `scaledBorrows = collateral` is not handled.

Recommendation

We recommend handling this missing case.

Alleviation

`[CertiK]`: The client changed the code so that `healAccount()` can be called when `1 >= percentage` in commit: 36feb1c4ae514e7cc8dbb581c5d40a0b0c86f949.

VPB-04 INTERFACES NOT INHERITED AND IMPROPER/MISSING IMPLEMENTATION

Category	Severity	Location	Status
Logical Issue	Medium	contracts/Lens/PoolLens.sol (Base): 149~150; contracts/Pool/PoolRegistry.sol (Base): 26, 342; contracts/Pool/PoolRegistryInterface.sol (Base): 13~14, 20; contracts/RiskFund/ProtocolShareReserve.sol (Base): 11; contracts/RiskFund/RiskFund.sol (Base): 17	Resolved

Description

The contract `PoolRegistry.sol` does not inherit its interface `PoolRegistryInterface.sol`. In addition, the function `getBookmarks()` is in `PoolRegistryInterface.sol`, however, the function is not implemented in `PoolRegistry.sol`. Furthermore, the function `getPoolsSupportedByAsset()` returns `uint256[]` in the interface, when the implementation returns an `address[]`. This also affects `getPoolsSupportedByAsset()` in `PoolLens.sol`, which also currently returns a `uint256[]`, when it should return an `address[]`.

The contract `RiskFund.sol` does not inherit its interface `IRiskFund.sol`.

The contract `ProtocolShareReserve.sol` does not inherit its interface `IProtocolShareReserve.sol`.

Recommendation

We recommend that the cited contracts inherit their corresponding interfaces to ensure any functions in the interface are implemented properly. In addition, we recommend updating the interfaces so that they return the proper values and only contain functions that have been implemented.

Alleviation

`[CertiK]`: The client made the recommended changes in commit f6584f2c3f297e5c899aaf51864cde9de7a20b9d.

VPB-05 | LACK OF STORAGE GAP IN UPGRADEABLE CONTRACT

Category	Severity	Location	Status
Logical Issue	Medium	contracts/Comptroller.sol (Base): 17; contracts/VToken.sol (Base): 19	Resolved

Description

There is no storage gap preserved in the following parent contracts of the cited logic contracts. Any logic contract that acts as a base contract, and which is inherited by other upgradeable child contracts should have a reasonably sized storage gap preserved to accommodate new state variables introduced by future upgrades.

`VToken`

- Inherited contract `VTokenStorage` should include a storage gap;
- Inherited contract `VTokenInterfaces` should include a storage gap.

`Comptroller`

- Inherited contract `ComptrollerV1Storage` should include a storage gap.

Recommendation

We recommend having a storage gap of a reasonable size preserved in the logic contract in case that new state variables are introduced in future upgrades. For more information, please refer to: https://docs.openzeppelin.com/contracts/3.x/upgradeable#storage_gaps.

Alleviation

`[CertiK]` : The client made the recommended changes in commits: 6350a92bce19f25fb0cfc448d9a512afe92ee621 and 905ad3c9161a02b31bbf6fb37a6df35d716db602.

VTV-02 | `mintBehalf()` CAN MINT TOKENS TO THE ZERO ADDRESS

Category	Severity	Location	Status
Logical Issue	Medium	contracts/VToken.sol (Base): 495	Resolved

Description

The `mintBehalf()` function allows a user to mint tokens on behalf of another user. However, there is no check that the input `minter`, the address to which the `vTokens` are minted, is not the zero address. This can cause issues if a user accidentally mints to the zero address as those `vTokens` will not be able to be recovered, causing the underlying to be locked in the contract. In addition, any supply rewards for that market can also then be claimed for the zero address, locking them forever.

Recommendation

We recommend checking that the input `minter` is not the zero address.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 014cd2463abd0a73414d501fabe18fb8a8d1e706.

COP-05 PRICE IS NOT UPDATED FOR ALL USERS MARKETS IN `preBorrowHook()`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/Comptroller.sol (update7): 340~341	Resolved

Description

The function `updatePrices()` calls `oracle_.updatePrice()` for all of the input accounts `accountAssets`. If the `preBorrowHook()` is called when the `borrower` is not currently entered in the market, then the market will be added to the borrowers `accountAssets` after `updatePrices()` is called. Thus it will not call `oracle_.updatePrice()` for the market that is being entered.

Recommendation

We recommend calling `updatePrices()` after the market has been entered and included in the borrowers `accountAssets`.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 108aa561f07115b8a571bfba92c89f6055b675c7.

PLV-01 | INCORRECT CALCULATION IN `PoolLens`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/Lens/PoolLens.sol (Addendum Base): 508~514	Resolved

Description

The function `calculateBorrowerReward()` returns the incorrect value if the case where users borrowed tokens before the market's borrow state index was set. In this case, it will always return 0, when it should return the rewards accumulated starting from the `rewardTokenInitialIndex`.

Recommendation

We recommend including logic similar to `calculateSupplierReward()` to account for this case.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 8f45885a322ebda9ee2e1204691d1d6500df1271.

RFR-01 | INEFFECTIVE CHECK FOR `minAmountToConvert`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/RiskFund/RiskFund.sol (Base): 218~221, 226~228	Acknowledged

Description

The variable `minAmountToConvert` is supposed to ensure an `underlyingAsset` is only swapped to the `convertibleBaseAsset` if the amount in USD exceeds it. In `_swapAsset()` an oracle is used to determine the expected value in USD of the `underlyingAsset`. However, the swap is always done using Pancakeswap, whose price can be manipulated. If the `amountOutMin` is set less than the `minAmountToConvert` a malicious user could manipulate the price on Pancakeswap and the assets swapped for less than the `minAmountToConvert`.

Recommendation

We recommend checking that the expected price in USD of the `amountOutMin` of the `convertibleBaseAsset` is greater than or equal to the `minAmountToConvert`. We also recommend carefully choosing minimum amounts to protect against price manipulations of the PancakeSwap pair.

Alleviation

`[CertiK]`: The client added the following check in commit: fa491ccd135126e2ad91609543e5e4ec3c212e5f

```
require(amountOutMin >= minAmountToConvert, "RiskFund: amountOutMin should be greater than minAmountToConvert");
```

This assumes that the `amountOutMin` is its amount in USD. However, this may not be the case depending on what `convertibleBaseAsset` is used and in the case it is pegged to USD it may be off peg when this function is called. We recommend using an oracle to look up the price of the `convertibleBaseAsset` in USD and calculating what the `amountOutMin` will be in USD, then verifying that amount is greater than or equal to the `minAmountToConvert`.

In addition, as the `minAmountToConvert` is greater than zero, the check above that `amountOutMin != 0` is redundant.

Furthermore, we recommend ensuring that the logic is compatible with the oracle and tokens that may have different decimals.

`[CertiK]`: The client acknowledged the issue and stated the following:

`[Venus]`: "Issue acknowledged. I will fix the issue in the future, which will not be included in this audit engagement. We'll review in the future how we'll swap assets to convertibleBaseAsset, probably not using PCS. We won't deploy RiskFund the day 0."

VPB-06 | MISSING ZERO ADDRESS VALIDATION

Category	Severity	Location	Status
Logical Issue	Minor	contracts/Comptroller.sol (Base): 165~166, 166~167, 1031; contracts/RiskFund/RiskFund.sol (Base): 72; contracts/Shortfall/Shortfall.sol (Base): 125~126; contracts/VToken.sol (Base): 86~87, 87~88, 88~89, 1357~1358	Resolved

Description

In `RiskFund.sol`:

The function `initialize()` does not check that the input `_accessControl` or `_shortFall` are not the zero address.

In `VToken.sol`:

- The function `_initialize()` does not check that the following inputs are not the zero address:
  - `admin_`
  - `address(accessControlManager)`
  - `riskManagement.shortfall`
  - `riskManagement.riskFund`
  - `riskManagement.protocolShareReserve`
- In the function `approve()` it is not checked that the input sender is not the zero address. It is expected that calls of the form `approve(spender, amount)` fail if the address in `spender` is the zero address;
- The function `setAccessControlAddress()` does not check that `newAccessControlManager` is the zero address.

In `Comptroller.sol`:

- In the `constructor()`, there is no check that the input addresses `poolRegistry_` and `accessControl_` are not the zero address;
- In function `setPriceOracle()`, there is no check that `address(newOracle)` is not `address(0)`.

In `Shortfall.sol`:

- The function `_initialize()` does not check that the following inputs are not the zero address:

- _convertibleBaseAsset
- riskFund

Recommendation

We recommend checking that the inputs are not the zero address.

Alleviation

`[CertiK]`: The client made the recommended changes in commits: 784a2f0f0d903c8708e1a353fd80d83b7c067506 and 4fdc72c4b5f6fd89737531402be42a7b87614b39.

VPB-08 | CHANGING `convertibleBaseAsset` CAN CAUSE ISSUES

Category	Severity	Location	Status
Logical Issue	Minor	contracts/RiskFund/RiskFund.sol (Base): 166; contracts/Shortfall/Shortfall.sol (Base): 133	Resolved

Description

In `RiskFund.sol`,

When `swapPoolsAssets()` is called, it swaps an underlying asset of a `vToken` to the convertible base asset and adds the amount of convertible base asset received to the `poolReserves` of the `vToken` 's associated pool. If an underlying asset is changed before all reserves are transferred for auction, then it is possible that the `poolReserves` value is accounting for two different tokens. This can cause confusion and require manual intervention to fix.

In `Shortfall.sol`,

The `convertibleBaseAsset` can be changed at any given time. This includes during an auction. However, the auction cannot be closed if the `convertibleBaseAsset` on `RiskFund` and `Shortfall` are different. If this occurs, it would require manual change to be able to close the auction.

Recommendation

In `RiskFund.sol`:

We recommend ensuring that if the `convertibleBaseAsset` is changed, that all pool reserves have been transferred for auction. Alternatively, functionality can be added to convert any of the old `convertibleBaseAsset` to the new `convertibleBaseAsset` and update the pool reserves accordingly.

In `Shortfall.sol`:

We recommend either locking the auction contract from being able to change `convertibleBaseAsset` while in process of auctioning off debt or add logic to hold the funds that are being auctioned if the value must be changed during an auction.

Alleviation

`[CertiK]`: The client resolved the issue by removing the functionality to change the `convertibleBaseAsset` and checking that the convertible base assets are the same when setting the `shortfallContractAddress` in commits: 7eb29738d2115bff840bc727a3c1f8af6f109996 and cc5c50098b8dedd6f04c9ae462e4fe20eb1b9b3d.

VPB-10 `initialize()` IS UNPROTECTED AND INCONSISTENT INIT FUNCTION

Category	Severity	Location	Status
control-flow	Minor	contracts/Rewards/RewardsDistributor.sol (Base): 90; contracts/RiskFund/ProtocolShareReserve.sol (Base): 33	Resolved

Description

The function `initialize()` in `RewardsDistributor.sol` is `public` and can be called by anyone as long as the contract is deployed.

The function `initialize()` in `ProtocolShareReserve.sol` calls `__Ownable_init()` instead of `__Ownable2Step_init()`.

Recommendation

We recommend adding the following to `RewardsDistributor`:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

The addition will prevent the function `initialize()` from being called directly in the implementation contract, but the proxy will still be able to `initialize()` its storage variables. It will also make this contract consistent with the other upgradeable contracts.

In addition, we recommend using `__Ownable2Step_init()` within the initializer function of contract `ProtocolShareReserve` to remain consistent. While currently the functionality of these are the same, if there are any updates to Openzeppelin's contracts this can cause issues with further deployments.

Alleviation

`[CertiK]`: The client made the recommended changes in commits: 26432394eefab2779944318f14df08e090cafde6 and 81909aa203ee2c78b0830a750268f72eb2af1d22.

VPH-01 | MISSING INPUT VALIDATION

Category	Severity	Location	Status
Volatile Code	Minor	contracts/Comptroller.sol (Base): 42~43, 45~46, 723~724, 796~797, 839, 839, 1031~1032; contracts/RiskFund/RiskFund.sol (Base): 163~164; contracts/Shortfall/Shortfall.sol (Base): 120, 300; contracts/MaxLoopsLimitHelper.sol (Addendum Base): 25~30	Resolved

Description

In the contract `Comptroller`, the following missing input validations were found:

- In function `setCloseFactor()`, there is no check to the update on `closeFactorMantissa`. There appear to be constant bounds that would be of use for this check.
- In function `setLiquidationIncentive()`, there is no check that the updated value `newLiquidationIncentiveMantissa` is a reasonable value for its intended use.
- Privileged function `setMaxLoopsLimit()` in the `Comptroller` can adjust the value `maxLoopsLimit` without making any checks against its previous value. If the `maxLoopsLimit` is updated to a value lower than its previous state, then some state array lengths may automatically exceed the new limit, with no way to reduce their size, while other array lengths may still remain below the threshold. We recommend ensuring that the `maxLoopsLimit` cannot be updated to a value that is less than a state array length that is checked against it.

In the contract `RiskFund`, the following missing input validations were found:

- Any address that has been given access to call `swapPoolsAssets()` by the access control manager, can input any array of `vToken` addresses to swap their pools assets reserves. However, these `vTokens` are never checked to ensure that the `comptroller` is registered and the `vToken` is a market of that comptroller.

In the contract `Shortfall`, the following missing input validations were found:

- In the initializer, the _minimumPoolBadDebt can be set to zero. We recommend not allowing this to be set to zero.
- The function `startAuction()` should check if the pool is registered.

Recommendation

We recommend performing the checks cited above.

Alleviation

`[CertiK]` : The client made the recommended changes in commits:

- 8ad300d1473347620df357fe6b98abc9f5ac4f98;
- 2772f545aa4f86e1e583091febed6cd07edf260d;
- 8ec437820647f825d390082fc935f8e6af2a31b1.

VTV-03 | APPROVAL RACE CONDITION

Category	Severity	Location	Status
Logical Issue	Minor	contracts/VToken.sol (Base): 218~219	Resolved

Description

The ERC-20 `approve` function has a well-known race condition whereby an adjustment of an existing approval can be exploited to utilize the full amount of the previously set approval as well as the newly set approval by monitoring the transaction mempool.

See also ERC20 API: An Attack Vector on Approve/TransferFrom Methods

Recommendation

We recommend including `increaseAllowance` and `decreaseAllowance` functions. For an example see OpenZeppelin's ERC20 contract.

Alleviation

`[CertiK]` : The client made the recommended changes in commits c2ab0fb7b786519ea9a625743c0d83a865e0147e.

COP-02 | POSSIBLE GAS ISSUE WITH `updatePrices()`

Category	Severity	Location	Status
Logical Issue	Informational	contracts/Comptroller.sol (update7): 1193~1206, 1275	Acknowledged

Description

The function `updatePrices()` calls `oracle_.updatePrice` for all of a users `accountAssets`. If a user has a large amount of assets added, then this may exceed the gas limit and cause a revert and prevent them from being liquidated. Note that the total number of markets supported by a comptroller is bounded above by the `maxLoopsLimit`, so we give this an informational severity.

Recommendation

We recommend ensuring the `maxLoopsLimit` is set to a low enough value to prevent a transaction from exceeding the gas limit.

Alleviation

`[CertiK]`: The client acknowledged the issue and stated the following:

`[Venus]`: We have analyzed the potential values of maxLoopsLimit and we'll start with a small number to guarantee the correctness of the protocol.

CVP-05 STORAGE VARIABLES DECLARED OUTSIDE OF `ComptrollerV1Storage`

Category	Severity	Location	Status
Coding Style	Informational	contracts/Comptroller.sol (Base): 18~19, 18~19, 24~25, 33~34, 39~40, 42~43, 45~46, 48~49, 52~53, 56~57, 59~60, 62~63	Resolved

Description

Storage variables are declared in the `Comptroller` contract; the `Comptroller` inherits from `ComptrollerV1Storage` and these variables should be declared within this child contract to maintain contract organization.

Recommendation

We recommend defining all storage variables for the `Comptroller` in the inherited `ComptrollerV1Storage` .

Alleviation

`[CertiK]` : The client made the recommended changes in commits:

- 0fd5c7e5e7a9c858d9e3733e1091967d4316d13f;
- 2cc427c8262666e43ffa10f963a1ee41bb79419c;
- 191c379560e3d916316d1ba76adb7a03df0d717d;
- 78c5a0947dda78e3df87ad16b6f2a4521ee12968;

JRM-01 | UNNECESSARY INHERITANCE

Category	Severity	Location	Status
Logical Issue	Informational	contracts/JumpRateModelV2.sol (Base): 5, 12	Resolved

Description

The contract `JumpRateModelV2` inherits both `InterestRateModel` and `BaseJumpRateModelV2`. However, `BaseJumpRateModelV2` also inherits `InterestRateModel` so that only `BaseJumpRateModelV2` needs to be inherited.

Recommendation

We recommend removing the unnecessary inheritance and import.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: bce488e8405d3491aba3dbd52805ab4ff02af651.

RDR-01 | ANYONE CAN CLAIM REWARDS FOR ANOTHER USER

Category	Severity	Location	Status
Logical Issue	Informational	contracts/Rewards/RewardsDistributor.sol (Base): 443, 452	Acknowledged

Description

The function `claimRewardToken()` can be called by anyone to claim the input `holder` rewards. However, a user could claim the rewards for a `holder` without their knowledge which could cause confusion if a holder tries to claim their rewards later and is expecting to receive a different value.

Recommendation

We recommend ensuring that a user cannot claim a `holder` 's rewards without their prior consent.

Alleviation

`[CertiK]` : The client acknowledged the issue, but decided to not make any changes to the current version.

RFR-02 | DUPLICATE IMPORTS

Category	Severity	Location	Status
Coding Style	Informational	contracts/RiskFund/RiskFund.sol (Base): 8, 11	Resolved

Description

In `RiskFund.sol`, the file `PoolRegistry.sol` is imported twice.

Recommendation

We recommend removing the duplicate import.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 465c0018874262f41cf3e29236ed2ebfaef7d8e8.

SSP-01 | INCONSISTENT COMMENT AND CODE

Category	Severity	Location	Status
Inconsistency	Informational	contracts/Shortfall/Shortfall.sol (Addendum Base): 161~171, 331	Resolved

Description

In the comment above `updateWaitForFirstBidder` it states: "If the first bid is not made within this limit, the auction is closed and needs to be restarted". However, this is not the case as the only time `waitForFirstBidder` is used is in a check when calling `restartAuction()`. It is not used in any check when calling `placeBid()` so that the bids can be placed after this limit.

Recommendation

If the comment reflects the intended functionality, we recommend only allowing bids to be placed within the specified limit.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 06e8ca20d279dc1e6233580b274c630b9ed85c79.

VPB-02 | INCOMPATIBILITY WITH FEE ON TRANSFER TOKENS

Category	Severity	Location	Status
Logical Issue	Informational	contracts/RiskFund/RiskFund.sol (Base): 236, 243; contracts/Shortfall/Shortfall.sol (Base): 170, 177, 180, 216~217, 219~220, 223, 229~230, 233~234; contracts/VToken.sol (Base): 736~737, 786~787, 984~985	Acknowledged

Description

The `VToken.sol` contract includes documentation on use with fee on transfer tokens as the `underlying`, and is intended to be compatible with such tokens. On the occasion where the `underlying` asset takes a fee on transfer, the update to `repayAmountFinal` in internal function `_repayBorrowFresh()` will prevent a user from paying down their full borrow balance in one transaction.

In contract `VToken.sol`, the `healBorrow()` function will call `_doTransferIn()` so the `payer` can repay a part of the borrows made by the borrower.
No check is made to ensure that `repayAmount == actualRepayAmount`, if a non standard token implementation is used as the underlying token, this might cause a borrow to be forgiven without it being repaid as much as it should.

In `RiskFund.sol`, the function `_swapAsset()` uses `swapExactTokensForTokens()`, which will revert if a token takes a fee on transfer. This will make it impossible for any `underlyingAsset` that takes a fee on transfer to be swapped to the `convertibleBaseAsset`. In addition, if the `convertibleBaseAsset` may be a fee on transfer token, then the `totalAmount` should be determined from the balance of the contract after the swap. This is because `amounts[1]` is the amount to be transferred before a fee is taken and will not be the actual amount received by the contract.

In `Shortfall`, if the tokens associated with a `Comptroller` have fees on transfer, `closeAuction()` and `placeBid()` could create a discrepancy between the badDebt and the amount of underlying token for the associated `VToken`. Indeed, there is no check to make sure that the input parameter of `badDebtRecovered()` matches the amount of token that has been effectively transferred.

Recommendation

For `vToken.sol`, we recommend changing the logic of `repayAmountFinal` to accommodate the possibility of fees on transfer.

We also recommend adding a check to prevent gaps between `repayAmount` and `actualRepayAmount` in `healBorrow()`.

For `RiskFund.sol`, we recommend instead using `swapExactTokensForTokensSupportingFeeOnTransferTokens()` which uses the actual received amount to calculate the output amount. Note that this may not work for all fee on transfer tokens as it is possible they are designed in a way that does not work with the router. Because of this, any asset that a market will be

created for should be carefully vetted to ensure that it interacts safely with the protocol. In addition, if the design intent is to allow a `convertibleBaseAsset` that takes a fee on transfer, we recommend checking the contracts balance after the swap to determine the `totalAmount`.

For `Shortfall.sol`, we recommend checking the balance after ERC20 tokens are transferred to ensure reverts do no occur.

We recommend adding a check to ensure that the amount of bad debt removed from a `VToken` has been paid.
For example `marketsDebt[i]` should be updated as the difference between the balances of the contract before and after the transfer to the `address(auction.markets[i])`, by doing so the removal of the bad debt will be consistent with the amount of ERC20 received by the `VToken` contract.

It should also be noted that they are common non-standard ERC20 implementations the protocol is not compatible with. We recommend creating a list of specifications which tokens used should be compliant with, to prevent any unexpected errors.

Alleviation

`[CertiK]`: The client stated they will not be using deflationary tokens in pools and that these types of tokens will be discarded when considering new tokens to add as markets. They acknowledged that the comments should be changed to avoid reference to these types of tokens and stated they will fix the issue in the future, which will not be included in this audit engagement.

VPB-11 | VOLATILE CONDITIONS

Category	Severity	Location	Status
Logical Issue	Informational	contracts/Comptroller.sol (Base): 617~618; contracts/Shortfall/Shortfall.sol (Base): 59, 201	Resolved

Description

There may be market conditions where a liquidator is incentivized to not liquidate a borrow, and instead call `healAccount()`. This is not desirable as it incurs bad debt for the protocol and liquidators should always be incentivized to liquidate as soon as possible. The possible impact of this depends on the value of `minLiquidatableCollateral`, as the larger the value, the more bad debt that can be incurred. As auctions require a minimum amount of bad debt to start, a liquidator may want to increase the bad debt so that an auction for the reserves occurs, betting that they will win the auction.

Please inform us of what the intended range of values for `minLiquidatableCollateral` are.

One of our concerns with `healAccount()` is the following scenario:

Assume for simplicity that the protocol does not take any fees. In general the amount of profit would be smaller as the cost of the price manipulation and protocol fees must be taken into account. However, if the token owner is able to mint their own tokens, the price can be manipulated for a low cost.

A user deposits 1000 USD of `tokenA` as collateral with a collateral factor of 80%. And then borrows 800 USD dollars of a stablecoin against it.

The user manipulates the price of tokenA causing the price to drop so low that the collateral is worth 100 USD.

The user then calls `healAccount()` on their own account paying a little less than 100 USD dollars to get approximately 100 USD in tokenA. (This is assuming the `minimumLiquidatableCollateral` is set to the highest provided estimated of 100 USD see the alleviation below.)

The price is then put back to normal so that the attacker has around 1000 USD dollars in tokenA and also has the 800 USD they borrowed originally. Minus the 100 USD they payed when healing the account this nets them around 700 USD at the cost of the Venus Protocol.

The user could repeat this process as many times as they wish, in general the more price manipulation and higher the collateral factor, the greater the amount that can be gained.

Recommendation

We recommend carefully considering these scenarios to ensure that the design of the protocol is resilient against all market conditions.

Alleviation

`[Venus]` : "Regarding `minLiquidatableCollateral` , the exact amount will depend on the gas price. Given the scenarios where it will be used (to fully liquidate accounts with a small amount of debt), we will set it in a range of 25-100 USD."

Venus then provided the following statement regarding the scenario outlined above:

"We shouldn't have a so manipulable market with a collateral factor of 80%. We follow Gauntlet recommendations to define the collateral factors, so if a market can be manipulated to decrease the underlying price by 90% so quickly that liquidators cannot liquidate that position before, it will probably have a collateral factor of 0% in the protocol."

`[CertiK]` : Assuming Gauntlet gives accurate recommendations this resolves the issue. However, this is out of the scope of this audit and when new tokens are added it should be ensured that Gauntlet will give an accurate recommendation for their collateral factor.

VPB-12 | MISSING OR UNUSED EMIT EVENTS

Category	Severity	Location	Status
Coding Style	Informational	contracts/Comptroller.sol (Base): 96~97, 617, 809~810, 827, 920, 920~921; contracts/Factories/VTokenProxyFactory.sol (Base): 26; contracts/Rewards/RewardsDistributor.sol (Base): 96, 328, 337, 358; contracts/VToken.sol (Base): 1410	Resolved

Description

There should always be events emitted in sensitive functions that are controlled by centralization roles.

In `ProtocolShareReserve.sol`, the following functions are missing emit events:

- `releaseFunds()`

In `Comptroller.sol`, the following functions are missing emit events:

- `addRewardsDistributor()`
- `supportMarket()`

The following events are declared but never used:

- `NewBorrowCapGuardian()`

In `RewardsDistributor.sol`, the following functions are missing emit events:

- `initializeMarket()`
- `updateRewardTokenSupplyIndex()`
- `_updateRewardTokenSupplyIndex()`
- `updateRewardTokenBorrowIndex()`
- `_updateRewardTokenSupplyIndex()`

In `VTokenProxyFactory.sol`, the following functions are missing emit events:

- `deployVTokenProxy()`

Recommendation

We recommend emitting events for the sensitive functions that are controlled by centralization roles.

Alleviation

`[CertiK]` : The client made the recommended change in the following commits:

- 2b34cd4f4c55166d2010eb7cb302da757a67f472;
- 595a6602d8e7267258500126a9eff31dc4890876;
- 2e9340f032ce7f645ffa5dc60b8e0fe23147d80c.

VPB-13 | MISSING ACCESS RESTRICTION

Category	Severity	Location	Status
control-flow	Informational	contracts/RiskFund/ReserveHelpers.sol (Base): 27~28; contracts/VToken.sol (Base): 1410~1411	Resolved

Description

Function `updateAssetsState()` can be called by anyone. A user can create a malicious `asset` ERC20 and make it appear as if the `ProtocolShareReserve` or `RiskFund` contracts have a balance of this malicious `asset` token. This will update the state of mappings `assetsReserves[asset]` and `poolAssetsReserves[comptroller][asset]` with the balance.

This is not a risk as long as the centralized authorities of the protocol do not add these `asset` addresses as markets in the corresponding `comptroller` addresses.

On a similar note, malicious tokens may be sent directly to `VToken` contracts. If `sweepToken()` is called on these malicious tokens, it may cause unforeseen actions to occur.

Recommendation

We recommend restricting access to `updateAssetsState()`. Moreover, we recommend carefully vetting all tokens transferred to a `VToken` contract before calling `sweepToken()` on them.

Alleviation

`[CertiK]`: The client resolved the issue by checking the asset is supported by the pool in commit: 478996af3175d987ee9154d76cdce46ce4721f61.

VPB-19 | ATYPICAL CONSTRUCTOR IMPLEMENTATION

Category	Severity	Location	Status
Coding Style	Informational	contracts/Comptroller.sol (Addendum Base): 123; contracts/ComptrollerStorage.sol (Addendum Base): 129~135	Resolved

Description

It is non-standard to include a `constructor` in an inherited construct which is used to house storage variables. We believe that this was to allow `immutable` variables to be included in `ComptrollerStorage`. However, `immutable` variables are not located in storage and are instead evaluated once at construction time and their value is copied to all the locations in the code where they are accessed.

Recommendation

We recommend using contract `ComptrollerStorage` to exclusively hold storage variables and moving the immutable variable to the `Comptroller` contract so that it can be assigned in the comptrollers `constructor`.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 422ac6df9b3183ee99c64194f20e1e90e9121b7b.

VPB-21 | ACCESS CONTROL CHANGES

Category	Severity	Location	Status
Inconsistency	Informational	contracts/Pool/PoolRegistry.sol (Addendum Base): 269, 272, 277~285, 345; contracts/RiskFund/ProtocolShareReserve.sol (Addendum Base): 70	Resolved

Description

In the contract `ProtocolShareReserve`

The `onlyOwner` modifier has been removed from `releaseFunds()`. This can cause funds to be released at unexpected times, as it can be called by any user. We want to ensure that this is the design intent.

In the contract `PoolRegistry`

When `addMarket()` is called the `initialSupply` will be taken from the `owner()` as opposed to the `msg.sender`. We want to ensure this is the design intent as previously this function could only be called by the owner so that necessarily the `msg.sender` was always the `owner()`.

Additionally, the function `setPoolName()` can only be called by an approved entity to change the pool name. However, this function does not perform the following check:

```
require(bytes(name).length <= 100, "Pool's name is too large.");
```

As this check is performed in `createRegistryPool()` which is also only callable by an approved entity, we would like to know if it is the design intent to allow `setPoolName()` to exceed this limit.

Recommendation

We recommend considering if the following is design intent.

Alleviation

`[CertiK]`: The client stated the changes to `ProtocolShareReserve` are by design. The client then added a check for the length of the pool name in `setPoolName()` in commit: 92de9024282fc783e7d0ea1445fb2657e3252de7. In addition, they changed `addMarket()` so that it is take from the `msg.sender` as opposed to the `owner()` in commit: fa4ffcc3e08562a0c385231032e8ab840f9a1e84.

VPH-02 | TYPOS AND INCONSISTENCIES

Category	Severity	Location	Status
Inconsistency	Informational	contracts/Comptroller.sol (Base): 422~424, 454, 472~473, 535~536, 1197; contracts/ComptrollerInterface.sol (Base): 80~81, 86~87, 90~91, 100~101; contracts/Factories/VTokenProxyFactory.sol (Base): 22; contracts/Lens/PoolLens.sol (Base): 252~253; contracts/Pool/PoolRegistry.sol (Base): 95, 116, 323, 357; contracts/Rewards/RewardsDistributor.sol (Base): 18~22, 26, 223~225, 259; contracts/RiskFund/ProtocolShareReserve.sol (Base): 26, 30; contracts/RiskFund/ReserveHelpers.sol (Base): 23; contracts/RiskFund/RiskFund.sol (Base): 35, 59, 73, 98, 103, 130, 133, 141, 145; contracts/Shortfall/Shortfall.sol (Base): 133, 137; contracts/VToken.sol (Base): 157, 163, 259, 476, 491, 563, 579, 671, 737, 752, 826, 1041, 1060; contracts/BaseJumpRateModelV2.sol (update7): 137; contracts/Shortfall/Shortfall.sol (update7): 229; contracts/WhitePaperInterestRateModel.sol (update7): 93; contracts/Comptroller.sol (Addendum Base): 171, 1176; contracts/Governance/AccessControlled.sol (Addendum Base): 58; contracts/Lens/PoolLens.sol (Addendum Base): 421; contracts/RiskFund/ProtocolShareReserve.sol (Addendum Base): 92~96	Resolved

Description

`RewardsDistributor`

The following typos and inconsistencies were found in this file:

- The comments for `_distributeSupplierRewardToken` are not above the function and instead above the mapping `rewardTokenSupplyState`;
- The `TODO` comment under `_distributeSupplierRewardToken` should either be deleted or explained and warned that it should only be called when a user is in the supplier market;
- The comment for `distributeBorrowerRewardToken()` mentions `Comptroller.borrowAllowed` and `Comptroller.repayBorrowAllowed` which are not functions defined in the comptroller. The comment should instead reference `Comptroller.preBorrowHook` and `Comptroller.preRepayHook`;
- The comment "vToken The market whose REWARD TOKEN speed to update" should be rephrased to be clearer.

`VToken`

The following typos and inconsistencies were found in this file:

- The word "compatilibily" is spelled incorrectly and should be corrected to "compatibility";
- The word "its" is incorrectly used and should be changed to "it's";
- The variable `srvTokensNew` should be `srcTokensNew`.

`PoolRegistry`

The following typos and inconsistencies were found in this file:

- In the comment for the `metadata` mapping, it says "Maps venus pool id to metadata", however, it maps a pool's `comptroller` address to metadata;
- In the comment for the `_vTokens` mapping it says "Maps pool id to asset to vToken.", however, it maps a pool's `comptroller` address to asset to `vToken`;
- In the comment for `getPoolByComptroller()` it says "The Comptroller implementation address", however, the address is the comptroller proxy address associated to the pool.

In addition, in the function `_registerPool()`, the check `bytes(name).length <= 100` reverts if the input `name` is too large. However, the error message states that no pool name was supplied. We recommend changing the error message or check to reflect one another.

`ProtocolShareReserve`

The following typos and inconsistencies were found in this file:

- In the comments above `initialize()`, `_protocolIncome` is the address the protocol income is sent to, not the remaining protocol income;
- In the function `initialize()`, the error message for the check that `_protocolIncome` is not the zero address does not reflect that it is the protocol's income address.

`RiskFund`

The following typos and inconsistencies were found in this file:

- In the comments for the function `initialize()`, it states "Asset should be worth of min amount to convert into base asset". We recommend changing this to something more clear such as "minimum amount assets must be worth to convert into base asset";
- In multiple locations, "amout" is spelled incorrectly and should be changed to "amount";
- In `swapPoolsAssets()`, the input array `underlyingAssets` can cause confusion as the addresses of the `vToken` of an underlying should be input, not the addresses of the underlying themselves. If this is changed, the comment for the parameter should also be changed to reflect it is an array of `vToken` addresses;

- The event `ConvertableBaseAssetUpdated` and function `setConvertableBaseAsset()` use "convertable", when it should be "convertible".

`ReserveHelpers`

The following typos and inconsistencies were found in this file:

- The comments for `updateAssetsState()` only mention updating "after transferring to risk fund", however, this function is also called when transferring funds to the protocol share reserve.

`Comptroller`

The following typos and inconsistencies were found in this file:

- The word "safelly" is spelled incorrectly and should be corrected to "safely";
- The function `preRepayHook()` does not use the inputs `payer` or `repayAmount`;
- The function `preLiquidateHook()` does not use the input `liquidator`;
- The function `preSeizeHook()` does not use the input seizeTokens;
- The comments above `preLiquidateHook()` state "Not restricted if vToken is enabled as collateral" which can cause confusion as the function is not restricted.

`ComptrollerInterface`

The following functions are not present within the `Comptroller` contract and can be removed from the `ComptrollerInterface`:

- `priceOracle()`
- `getXVSRewardsByMarket()`
- `compSpeeds()`

`VTokenProxyFactory`

The address `vTokenProxyAdmin_` is never used when deploying the `vToken`. We recommend either implementing or removing this variable from the `VTokenArgs` struct.

`Shortfall`

- The function `setConvertableBaseAsset()` is spelled incorrectly and should be spelled `setConvertibleBaseAsset()`.

- The emitted event `ConvertableBaseAssetUpdated` is spelled incorrectly and should be spelled `ConvertibleBaseAssetUpdated`.
- The event `ConvertableBaseAssetUpdated` is spelled incorrectly and should be spelled `ConvertibleBaseAssetUpdated`

The following typos and inconsistencies were found in commit: 78c5a0947dda78e3df87ad16b6f2a4521ee12968:

AccessControlled

The following typos and inconsistencies were found in this file:

- The error message in `setAccessControlManager()` has "acess" instead of "access".

ProtocolShareReserve

The following typos and inconsistencies were found in this file:

- The comment above `updateAssetsState()` states "@dev Update the reserve of the asset for the specific pool after transferring to risk fund." However, this should reference when transferring to the protocol share reserve as opposed to the risk fund.

PoolLens

The following typos and inconsistencies were found in this file:

- In comment for the function `_calculateNotDistributedAwards()`, "oreder" is used as opposed to "order".

BaseJumpRateModelV2

The following typos and inconsistencies were found in this file:

- The comment above `updateJumpRateModel()` states "(only callable by owner, i.e. Timelock)", however, it is callable by anyone who has been allowed to call it through the access control manager.

`Comptroller.sol`

- Within the comments for function `exitMarket()`, the word "disabling" is misspelled as "disabeling";
- Within the comments for function `setMaxLoopsLimit()`, the `@notice` reads "Set the limit for the loops can iterate to avoid the DOS". This comment may read more clearly as "Set the for loop iteration limit to avoid DOS";

The following typos and inconsistencies were found in commit: 74d3d384225da763e6fcecc688e0f06576016b33:

In the contract `Shortfall` :

The functions `closeAuction()` and `restartAuction` have the comment "@custom:event Errors if auctions are paused". However, these will not revert in this case. We recommend determining if these functions should be able to be called when paused and either removing the comment or adding a check to enforce the comment.

In the contract `BaseJumpRateModelV2` :

The comment "// Utilization rate is 0 when there are no borrows" does not reflect that `badDebt` must also be zero for this to be the case.

In the contract `WhitePaperInterstRateModel` :

The comment "// Utilization rate is 0 when there are no borrows" does not reflect that `badDebt` must also be zero for this to be the case.

Recommendation

We recommend fixing these typos and inconsistencies to improve clarity and readability.

Alleviation

`[CertiK]` : The client made the recommended changes in the following commits:

- 0da0aed3c48cd9422f80679b5ba34fd75131b2c2;
- 78f956a53b1973a7a98a57e5905584654ad03b75;
- 36b8fa6b66ed85c344ccf3be63ed3321aecbf64a;
- f2977e0b033104055ed758e38d57bcca189b640e;

VTV-04 | UNNECESSARY REFERENCE TO `VTokenInterface`

Category	Severity	Location	Status
Coding Style	Informational	contracts/VToken.sol (Base): 88~89	Resolved

Description

Since contract `VToken` inherits from `VTokenInterface`, it is unnecessary to refer to this contract when referencing struct type `RiskManagementInit`.

Recommendation

We recommend removing the unnecessary reference to `VTokenInterface` from the parameter.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 1b5291f372d30388c63cbe002c2bf33b171fe352.

OPTIMIZATIONS | VENUS - ISOLATED POOLS

ID	Title	Category	Severity	Status
COP-04	Unnecessary And Incorrect Check	Logical Issue	Optimization	Resolved
PRV-04	Unnecessary Parameter	Coding Style	Optimization	Resolved
RDR-02	Missing/Unnecessary Functionality	Gas Optimization	Optimization	Resolved
VPB-15	Unused State Variable	Gas Optimization	Optimization	Resolved
VPB-16	Unneccessary Use Of Payable Casting	Coding Style	Optimization	Resolved
VPB-22	User-Defined Getters	Coding Style	Optimization	Resolved
VTV-05	Inefficient Memory/Storage Parameters	Gas Optimization	Optimization	Resolved
WPM-01	Variables That Could Be Declared As Immutable	Gas Optimization	Optimization	Resolved

COP-04 | UNNECESSARY AND INCORRECT CHECK

Category	Severity	Location	Status
Logical Issue	Optimization	contracts/Comptroller.sol (update7): 350, 832, 836, 1240~1242, 1276	Resolved

Description

It is checked that the `msg.sender` is the `vToken`, so `accountAssets[msg.sender]` will not be the account assets of the `borrower`.

However, this check is unnecessary as other checks prevent the account assets from exceeding the `maxLoopsLimit`. In particular note that whenever a market is supported, via `supportMarket()`, when `_addMarket()` is called it ensures that after adding the market to the `allMarkets` array it does not exceed the `maxLoopsLimit`. As the only way for a market to be listed is through calling `supportMarket()`, this ensures that the total number of listed markets does not exceed the `maxLoopsLimit`.

The function `_addToMarket()` will only allow borrowers to be added to markets that are listed. As the listed markets do not exceed the `maxLoopsLimit` it follows that a borrower cannot be added to enough markets that it would exceed the `maxLoopsLimit`.

Recommendation

We recommend removing the unnecessary check.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 22b124317d8f7313c9135baf0a01f2e20718054f.

PRV-04 | UNNECESSARY PARAMETER

Category	Severity	Location	Status
Coding Style	Optimization	contracts/Pool/PoolRegistry.sol (Addendum Base): 48	Resolved

Description

The struct `AddMarketInput` has the parameter `vTokenProxyAdmin`, however, it is never used.

Recommendation

We recommend removing or implementing the unused parameter.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 2fd7018929fb9f529051aa3bdf0a236906c1cbb9.

RDR-02 | MISSING/UNNECESSARY FUNCTIONALITY

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/Rewards/RewardsDistributor.sol (Base): 409~437	Resolved

Description

The internal function `_claimRewardToken` takes an array of addresses `holders` as input as well as two bools `borrowers` and `suppliers`. It is only called in `RewardsDistributor.sol` by `claimRewardToken()`, with a single holder input and both bools always set to true. With the current design the inputs of `_claimRewardToken()` can be changed to accept a single address `holder` and omit the bool inputs.

Recommendation

We recommend either implementing functionality that makes use of the inputs or to change the inputs to match the design intent.

Alleviation

`[CertiK]`: The client made the recommended changes in commits: 2a37cad21f0c3b627f8391beeb6447428eb8b2c9 and b3d53030012993270beaf7fa074763c64e5f7645.

VPB-15 | UNUSED STATE VARIABLE

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/ComptrollerInterface.sol (Base): 88~89, 94~95; contracts/ComptrollerStorage.sol (Base): 26~27, 61~65, 61~62, 70~72; contracts/Lens/PoolLens.sol (Base): 30~31, 238~239	Resolved

Description

The following state variables are never used in the `Comptroller` codebase:

```
uint256 public maxAssets;
```

```
address public pauseGuardian;
bool public transferGuardianPaused;
bool public seizeGuardianPaused;
mapping(address => bool) public mintGuardianPaused;
mapping(address => bool) public borrowGuardianPaused;
```

```
address public borrowCapGuardian;
```

Recommendation

We recommend removing the unused variables and all references to them within other contracts.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: c0f7dcb11f9edd32503c5e9c116fc7fba7f19bfa.

VPB-16 | UNNECCESSARY USE OF PAYABLE CASTING

Category	Severity	Location	Status
Coding Style	Optimization	contracts/Factories/VTokenProxyFactory.sol (Base): 22~23; contracts/Lens/PoolLens.sol (Base): 79~80, 177~178; contracts/Pool/PoolRegistry.sol (Base): 256~257; contracts/VToken.sol (Base): 49~50, 86~87, 530~531, 571~572, 584~585, 595, 657~658, 679~680, 687~688, 724~725, 794~795, 1446~1447	Resolved

Description

The `payable` casting is used within the `VToken` contract, but this contract is not set up to be compatible with ETH.

Recommendation

We recommend removing the `payable` casting and removing mentions of ETH in the corresponding comments in which this casting is used.

Alleviation

`[CertiK]`: The client made the recommended changes in commits:

- 6cc1b726615207cada5afacfe4e7f09f77148447;
- a7de78e28ffa2a9576008b2925045d4c1b91468e;
- b0b4d83ca5003807e4b1b3ec5edae8c693a47af.

VPB-22 | USER-DEFINED GETTERS

Category	Severity	Location	Status
Coding Style	Optimization	contracts/Pool/PoolRegistry.sol (Addendum Base): 322~324; contracts/RiskFund/RiskFund.sol (Addendum Base): 209~216	Resolved

Description

`poolReserves` is public and thus has a compiler-generated getter function making the `getPoolReserve()` function unnecessary.

Recommendation

We recommend removing `getPoolReserve()` as it is equivalent to the compiler-generated getter function for `poolReserves`.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 552755b6e85f76328569e77f6d6a8440195cb287.

VTV-05 | INEFFICIENT MEMORY/STORAGE PARAMETERS

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/VToken.sol (Base): 337	Resolved

Description

In `VToken.sol`, the function `_borrowBalanceStored()` is a `view` function using `storage` to read contents from a structure.

In `AccessControlManager.sol`, the functions `isAllowedToCall()`, `hasPermission()`, `giveCallPermission()`, and `revokeCallPermission()` all have the input parameter `functionSig` with a data location of `memory`. The `functionSig` is never modified and is only used to compute the hash for each role. In addition, the functions are never called internally within the contract. Thus, their data location can be changed to `calldata` to avoid the gas consumption copying from `calldata` to `memory`.

Recommendation

We recommend using `memory` instead of `storage` for the function in `VToken.sol` and using `calldata` instead of `memory` for the functions mentioned in `AccessControlManager.sol`.

Alleviation

`[CertiK]`: The client made the recommended changes in the following commits:

- 9455ac037bdae6386761bc1a7d5b54261eb225e5;
- bedc3807c6435c95557c4380efdb9aa08a2d4ae9;
- 39a1a17c0af2d0c15fc83fb7c88d8a346e450f41.

WPM-01 | VARIABLES THAT COULD BE DECLARED AS IMMUTABLE

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/WhitePaperInterestRateModel.sol (Addendum Base): 22, 27	Resolved

Description

The linked variables assigned in the constructor can be declared as `immutable`. Immutable state variables can be assigned during contract creation but will remain constant throughout the lifetime of a deployed contract. A big advantage of immutable variables is that reading them is significantly cheaper than reading from regular state variables since they will not be stored in storage.

Recommendation

We recommend declaring these variables as immutable. Please note that the `immutable` keyword only works in Solidity version `v0.6.5` and up.

Alleviation

`[CertiK]`: The client made the recommended changes in commit: 13b081f69a17fb3ac00f6493e7e02f2c1b8a6bae.

FORMAL VERIFICATION | VENUS - ISOLATED POOLS

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied automated formal verification (symbolic model checking) to prove that well-known functions in the smart contracts adhere to their expected behavior.

Considered Functions And Scope

In the following, we provide a description of the properties that have been used in this audit. They are grouped according to the type of contract they apply to.

Verification of ERC-20 Compliance

We verified properties of the public interface of those token contracts that implement the ERC-20 interface. This covers

- Functions `transfer` and `transferFrom` that are widely used for token transfers,
- functions `approve` and `allowance` that enable the owner of an account to delegate a certain subset of her tokens to another account (i.e. to grant an allowance), and
- the functions `balanceOf` and `totalSupply`, which are verified to correctly reflect the internal state of the contract.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
erc20-transfer-revert-zero	Function `transfer` Prevents Transfers to the Zero Address
erc20-transfer-succeed-normal	Function `transfer` Succeeds on Admissible Non-self Transfers
erc20-transfer-succeed-self	Function `transfer` Succeeds on Admissible Self Transfers
erc20-transfer-correct-amount	Function `transfer` Transfers the Correct Amount in Non-self Transfers
erc20-transfer-correct-amount-self	Function `transfer` Transfers the Correct Amount in Self Transfers
erc20-transfer-change-state	Function `transfer` Has No Unexpected State Changes
erc20-transfer-exceed-balance	Function `transfer` Fails if Requested Amount Exceeds Available Balance
erc20-transfer-recipient-overflow	Function `transfer` Prevents Overflows in the Recipient's Balance
erc20-transfer-false	If Function `transfer` Returns `false`, the Contract State Has Not Been Changed
erc20-transfer-never-return-false	Function `transfer` Never Returns `false`

Property Name	Title
erc20-transferfrom-revert-from-zero	Function `transferFrom` Fails for Transfers From the Zero Address
erc20-transferfrom-revert-to-zero	Function `transferFrom` Fails for Transfers To the Zero Address
erc20-transferfrom-succeed-normal	Function `transferFrom` Succeeds on Admissible Non-self Transfers
erc20-transferfrom-succeed-self	Function `transferFrom` Succeeds on Admissible Self Transfers
erc20-transferfrom-correct-amount	Function `transferFrom` Transfers the Correct Amount in Non-self Transfers
erc20-transferfrom-correct-amount-self	Function `transferFrom` Performs Self Transfers Correctly
erc20-transferfrom-correct-allowance	Function `transferFrom` Updated the Allowance Correctly
erc20-transferfrom-change-state	Function `transferFrom` Has No Unexpected State Changes
erc20-transferfrom-fail-exceed-balance	Function `transferFrom` Fails if the Requested Amount Exceeds the Available Balance
erc20-transferfrom-fail-exceed-allowance	Function `transferFrom` Fails if the Requested Amount Exceeds the Available Allowance
erc20-transferfrom-fail-recipient-overflow	Function `transferFrom` Prevents Overflows in the Recipient's Balance
erc20-transferfrom-false	If Function `transferFrom` Returns `false`, the Contract's State Has Not Been Changed
erc20-transferfrom-never-return-false	Function `transferFrom` Never Returns `false`
erc20-totalsupply-succeed-always	Function `totalSupply` Always Succeeds
erc20-totalsupply-correct-value	Function `totalSupply` Returns the Value of the Corresponding State Variable
erc20-totalsupply-change-state	Function `totalSupply` Does Not Change the Contract's State
erc20-balanceof-succeed-always	Function `balanceOf` Always Succeeds
erc20-balanceof-correct-value	Function `balanceOf` Returns the Correct Value
erc20-balanceof-change-state	Function `balanceOf` Does Not Change the Contract's State
erc20-allowance-succeed-always	Function `allowance` Always Succeeds
erc20-allowance-correct-value	Function `allowance` Returns Correct Value

Property Name	Title
erc20-allowance-change-state	Function `allowance` Does Not Change the Contract's State
erc20-approve-revert-zero	Function `approve` Prevents Giving Approvals For the Zero Address
erc20-approve-succeed-normal	Function `approve` Succeeds for Admissible Inputs
erc20-approve-correct-amount	Function `approve` Updates the Approval Mapping Correctly
erc20-approve-change-state	Function `approve` Has No Unexpected State Changes
erc20-approve-false	If Function `approve` Returns `false`, the Contract's State Has Not Been Changed
erc20-approve-never-return-false	Function `approve` Never Returns `false`

Verification Results

In the remainder of this section, we list all contracts where model checking of at least one property was not successful. There are several reasons why this could happen:

- Model checking reports a counterexample that violates the property. Depending on the counterexample,this occurs if
    - The specification of the property is too generic and does not accurately capture the intended behavior of the smart contract. In that case, the counterexample does not indicate a problem in the underlying smart contract. We report such instances as being "inapplicable".
    - The property is applicable to the smart contract. In that case, the counterexample showcases a problem in the smart contract and a correspond finding is reported separately in the Findings section of this report. In the following tables, we report such instances as "invalid". The distinction between spurious and actual counterexamples is done manually by the auditors.
- The model checking result is inconclusive. Such a result does not indicate a problem in the underlying smart contract. An inconclusive result may occur if
    - The model checking engine fails to construct a proof. This can happen if the logical deductions necessary are beyond the capabilities of the automated reasoning tool. It is a technical limitation of all proof engines and cannot be avoided in general.
    - The model checking engine runs out of time or memory and did not produce a result. This can happen if automatic abstraction techniques are ineffective or of the state space is too big.

Detailed Results For Contract VToken (contracts/VToken.sol) In Commit 1ec61863d0d498a16c3fb4da2dc15021b06c96a6

Verification of ERC-20 Compliance

Detailed results for function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-revert-zero	Inconclusive	
erc20-transfer-succeed-normal	Inconclusive	
erc20-transfer-succeed-self	Inconclusive	
erc20-transfer-correct-amount	Inconclusive	
erc20-transfer-correct-amount-self	Inconclusive	
erc20-transfer-change-state	Inconclusive	
erc20-transfer-exceed-balance	Inconclusive	
erc20-transfer-recipient-overflow	Inconclusive	
erc20-transfer-false	Inconclusive	
erc20-transfer-never-return-false	Inconclusive	

Detailed results for function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-revert-from-zero	Inconclusive	
erc20-transferfrom-revert-to-zero	Inconclusive	
erc20-transferfrom-succeed-normal	Inconclusive	
erc20-transferfrom-succeed-self	Inconclusive	
erc20-transferfrom-correct-amount	Inconclusive	
erc20-transferfrom-correct-amount-self	Inconclusive	
erc20-transferfrom-correct-allowance	Inconclusive	
erc20-transferfrom-change-state	Inconclusive	
erc20-transferfrom-fail-exceed-balance	Inconclusive	
erc20-transferfrom-fail-exceed-allowance	Inconclusive	
erc20-transferfrom-fail-recipient-overflow	Inconclusive	
erc20-transferfrom-false	Inconclusive	
erc20-transferfrom-never-return-false	Inconclusive	

Detailed results for function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-succeed-always	True	
erc20-totalsupply-correct-value	True	
erc20-totalsupply-change-state	True	

Detailed results for function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-succeed-always	True	
erc20-balanceof-correct-value	True	
erc20-balanceof-change-state	True	

Detailed results for function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-succeed-always	True	
erc20-allowance-correct-value	True	
erc20-allowance-change-state	True	

Detailed results for function `approve`

Property Name	Final Result	Remarks
erc20-approve-revert-zero	False	Client Made Recommended Changes
erc20-approve-succeed-normal	True	
erc20-approve-correct-amount	True	
erc20-approve-change-state	True	
erc20-approve-false	True	
erc20-approve-never-return-false	True	

APPENDIX | VENUS - ISOLATED POOLS

Finding Categories

Categories	Description
Centralization	Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.
Gas Optimization	Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.
Logical Issue	Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.
	Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.
Coding Style	Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.
Inconsistency	Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

Details on Formal Verification

Technical description

Some Solidity smart contracts from this project have been formally verified using symbolic model checking. Each such contract was compiled into a mathematical model which reflects all its possible behaviors with respect to the property. The model takes into account the semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

The model also formalizes a simplified execution environment of the Ethereum blockchain and a verification harness that performs the initialization of the contract and all possible interactions with the contract. Initially, the contract state is initialized non-deterministically (i.e. by arbitrary values) and over-approximates the reachable state space of the contract throughout any actual deployment on chain. All valid results thus carry over to the contract's behavior in arbitrary states after it has been deployed.

Assumptions and simplifications

The following assumptions and simplifications apply to our model:

- Gas consumption is not taken into account, i.e. we assume that executions do not terminate prematurely because they run out of gas.
- The contract's state variables are non-deterministically initialized before invocation of any of those functions. That ignores contract invariants and may lead to false positives. It is, however, a safe over-approximation.
- The verification engine reasons about unbounded integers. Machine arithmetic is modeled as operations on the congruence classes arising from the bit-width of the underlying numeric type. This ensures that over- and underflow characteristics are faithfully represented.
- Certain low-level calls and inline assembly are not supported and may lead to an ERC-20 token contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

Formalism for property definitions

All properties are expressed in linear temporal logic (LTL). For that matter, we treat each invocation of and each return from a public or an external function as a discrete time steps. Our analysis reasons about the contract's state upon entering and upon leaving public or external functions.

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates to reason about the validity of atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `started(f, [cond])` Indicates an invocation of contract function `f` within a state satisfying formula `cond`.
- `willSucceed(f, [cond])` Indicates an invocation of contract function `f` within a state satisfying formula `cond` and considers only those executions that do not revert.
- `finished(f, [cond])` Indicates that execution returns from contract function `f` in a state satisfying formula `cond`. Here, formula `cond` may refer to the contract's state variables and to the value they had upon entering the function (using the `old` function).
- `reverted(f, [cond])` Indicates that execution of contract function `f` was interrupted by an exception in a contract state satisfying formula `cond`.

The verification performed in this audit operates on a harness that non-deterministically invokes a function of the contract's public or external interface. All formulas are analyzed w.r.t. the trace that corresponds to this function invocation.

Description of ERC-20 Properties

The specifications are designed such that they capture the desired and admissible behaviors of the ERC-20 functions `transfer`, `transferFrom`, `approve`, `allowance`, `balanceOf`, and `totalSupply`.

In the following, we list those property specifications.

Properties for ERC-20 function `transfer`

erc20-transfer-revert-zero

Function `transfer` Prevents Transfers to the Zero Address.

Any call of the form `transfer(recipient, amount)` must fail if the recipient address is the zero address.

Specification:

```
[](started(contract.transfer(to, value), to == address(0))
   ==> <>(reverted(contract.transfer) || finished(contract.transfer(to, value),
          !return)))
```

erc20-transfer-succeed-normal

Function `transfer` Succeeds on Admissible Non-self Transfers.

All invocations of the form `transfer(recipient, amount)` must succeed and return `true` if

- the `recipient` address is not the zero address,
- `amount` does not exceed the balance of address `msg.sender`,
- transferring `amount` to the `recipient` address does not lead to an overflow of the recipient's balance, and
- the supplied gas suffices to complete the call.

Specification:

```
[](started(contract.transfer(to, value), to != address(0)
          && to != msg.sender && value >= 0 && value <= _balances[msg.sender]
          && _balances[to] + value <= type(uint256).max && _balances[to] >= 0
          && _balances[msg.sender] <= type(uint256).max)
          ==> <>(finished(contract.transfer(to, value), return)))
```

erc20-transfer-succeed-self

Function `transfer` Succeeds on Admissible Self Transfers.

All self-transfers, i.e. invocations of the form `transfer(recipient, amount)` where the `recipient` address equals the address in `msg.sender` must succeed and return `true` if

- the value in `amount` does not exceed the balance of `msg.sender` and
- the supplied gas suffices to complete the call.

Specification:

```
[](started(contract.transfer(to, value), to != address(0)
          && to == msg.sender && value >= 0 && value <= _balances[msg.sender]
          && _balances[msg.sender] >= 0
          && _balances[msg.sender] <= type(uint256).max
          ==> <>(finished(contract.transfer(to, value), return)))
```

erc20-transfer-correct-amount

Function `transfer` Transfers the Correct Amount in Non-self Transfers.

All non-reverting invocations of `transfer(recipient, amount)` that return `true` must subtract the value in `amount` from the balance of `msg.sender` and add the same value to the balance of the `recipient` address.

Specification:

```
[](willSucceed(contract.transfer(to, value), to != msg.sender
   && _balances[to] >= 0 && value >= 0
   && _balances[to] + value <= type(uint256).max
   && _balances[msg.sender] >= 0 && _balances[msg.sender] <= type(uint256).max)
      ==> <>(finished(contract.transfer(to, value), return
             ==> _balances[msg.sender] == old(_balances[msg.sender]) - value
                 && _balances[to] == old(_balances[to]) + value)))
```

erc20-transfer-correct-amount-self

Function `transfer` Transfers the Correct Amount in Self Transfers.

All non-reverting invocations of `transfer(recipient, amount)` that return `true` and where the `recipient` address equals `msg.sender` (i.e. self-transfers) must not change the balance of address `msg.sender`.

Specification:

```
[](willSucceed(contract.transfer(to, value), to == msg.sender
  && _balances[to] >= 0 && _balances[to] <= type(uint256).max)
     ==> <>(finished(contract.transfer(to, value), return
         ==> _balances[to] == old(_balances[to]))))
```

erc20-transfer-change-state

Function `transfer` Has No Unexpected State Changes.

All non-reverting invocations of `transfer(recipient, amount)` that return `true` must only modify the balance entries of the `msg.sender` and the `recipient` addresses.

Specification:

```
[](willSucceed(contract.transfer(to, value), p1 != msg.sender && p1 != to)
  ==> <>(finished(contract.transfer(to, value), return
      ==> (_totalSupply == old(_totalSupply) && _allowances == old(_allowances)
           && _balances[p1] == old(_balances[p1])  ))))
```

erc20-transfer-exceed-balance

Function `transfer` Fails if Requested Amount Exceeds Available Balance.

Any transfer of an amount of tokens that exceeds the balance of `msg.sender` must fail.

Specification:

```
[](started(contract.transfer(to, value), value > _balances[msg.sender]
   && _balances[msg.sender] >= 0 && value <= type(uint256).max)
   ==> <>(reverted(contract.transfer) || finished(contract.transfer(to, value),
          !return)))
```

erc20-transfer-recipient-overflow

Function `transfer` Prevents Overflows in the Recipient's Balance.

Any invocation of `transfer(recipient, amount)` must fail if it causes the balance of the `recipient` address to overflow.

Specification:

```
[](started(contract.transfer(to, value), to != msg.sender
   && _balances[to] + value > type(uint256).max
   && _balances[to] >= 0 && _balances[to] <= type(uint256).max
   && _balances[msg.sender] <= type(uint256).max
   && value > 0 && value <= _balances[msg.sender])
   ==> <>(reverted(contract.transfer) || finished(contract.transfer(to, value),
          !return) || finished(contract.transfer(to, value), _balances[to]
                      > old(_balances[to]) + value - type(uint256).max - 1)))
```

erc20-transfer-false

If Function `transfer` Returns `false` , the Contract State Has Not Been Changed.

If the `transfer` function in contract `contract` fails by returning `false` , it must undo all state changes it incurred before returning to the caller.

Specification:

```
[](willSucceed(contract.transfer(to, value))
   ==> <>(finished(contract.transfer(to, value), !return]
   ==> (_balances == old(_balances) && _totalSupply == old(_totalSupply)
                                    && _allowances == old(_allowances)  ))))
```

erc20-transfer-never-return-false

Function `transfe` Never Returns `false`.

The transfer function must never return `false` to signal a failure.

Specification:

```
[](!(finished(contract.transfer, !return)))
```

Properties for ERC-20 function `transferFrom`

erc20-transferfrom-revert-from-zero

Function `transferFrom` Fails for Transfers From the Zero Address.

All calls of the form `transferFrom(from, dest, amount)` where the `from` address is zero, must fail.

Specification:

```
[](started(contract.transferFrom(from, to, value), from == address(0))
   ==> <>(reverted(contract.transferFrom) || finished(contract.transferFrom,
          !return)))
```

erc20-transferfrom-revert-to-zero

Function `transferFrom` Fails for Transfers To the Zero Address.

All calls of the form `transferFrom(from, dest, amount)` where the `dest` address is zero, must fail.

Specification:

```
[](started(contract.transferFrom(from, to, value), to == address(0))
   ==> <>(reverted(contract.transferFrom) || finished(contract.transferFrom,
          !return)))
```

erc20-transferfrom-succeed-normal

Function `transferFrom` Succeeds on Admissible Non-self Transfers. All invocations of `transferFrom(from, dest, amount)` must succeed and return `true` if

- the value of `amount` does not exceed the balance of address `from`,

- the value of `amount` does not exceed the allowance of `msg.sender` for address `from`,
- transferring a value of `amount` to the address in `dest` does not lead to an overflow of the recipient's balance, and
- the supplied gas suffices to complete the call.

Specification:

```
[](started(contract.transferFrom(from, to, value), from != address(0)
   && to != address(0) && from != to && value <= _balances[from]
   && value <= _allowances[from][msg.sender]
   && _balances[to] + value <= type(uint256).max
   && value >= 0 && _balances[to] >= 0 && _balances[from] >= 0
   && _balances[from] <= type(uint256).max
   && _allowances[from][msg.sender] >= 0
   && _allowances[from][msg.sender] <= type(uint256).max)
      ==> <>(finished(contract.transferFrom(from, to, value), return)))
```

erc20-transferfrom-succeed-self

Function `transferFrom` Succeeds on Admissible Self Transfers.

All invocations of `transferFrom(from, dest, amount)` where the `dest` address equals the `from` address (i.e. self-transfers) must succeed and return `true` if:

- The value of `amount` does not exceed the balance of address `from`,
- the value of `amount` does not exceed the allowance of `msg.sender` for address `from`, and
- the supplied gas suffices to complete the call.

Specification:

```
[](started(contract.transferFrom(from, to, value), from != address(0)
   && from == to && value <= _balances[from]
   && value <= _allowances[from][msg.sender]
   && value >= 0 && _balances[from] <= type(uint256).max
   && _allowances[from][msg.sender] <= type(uint256).max)
     ==> <>(finished(contract.transferFrom(from, to, value), return)))
```

erc20-transferfrom-correct-amount

Function `transferFrom` Transfers the Correct Amount in Non-self Transfers.

All invocations of `transferFrom(from, dest, amount)` that succeed and that return `true` subtract the value in `amount` from the balance of address `from` and add the same value to the balance of address `dest`.

Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), from != to && value >= 0
&& _balances[from] >= 0 && _balances[from] <= type(uint256).max
&& _balances[to] >= 0 && _balances[to] + value <= type(uint256).max)
   ==> <>(finished(contract.transferFrom(from, to, value), return
          ==> _balances[from] == old(_balances[from]) - value
              && _balances[to] == old(_balances[to] + value))))
```

erc20-transferfrom-correct-amount-self

Function `transferFrom` Performs Self Transfers Correctly.

All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` and where the address in `from` equals the address in `dest` (i.e. self-transfers) do not change the balance entry of the `from` address (which equals `dest`).

Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), from == to
   && value >= 0 && value <= type(uint256).max && _balances[from] >= 0
   && _balances[from] <= type(uint256).max)
      ==> <>(finished(contract.transferFrom(from, to, value), return
             ==> _balances[from] == old(_balances[from]))))
```

erc20-transferfrom-correct-allowance

Function `transferFrom` Updated the Allowance Correctly.

All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` must decrease the allowance for address `msg.sender` over address `from` by the value in `amount`.

Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), value >= 0
   && value <= type(uint256).max && _balances[from] >= 0
   && _balances[from] <= type(uint256).max && _balances[to] >= 0
   && _balances[to] <= type(uint256).max && _allowances[from][msg.sender] >= 0
   && _allowances[from][msg.sender] <= type(uint256).max)
      ==> <>(finished(contract.transferFrom(from, to, value), return
          ==> ((_allowances[from][msg.sender]
                == old(_allowances[from][msg.sender]) - value)
               || (_allowances[from][msg.sender]
                   == old(_allowances[from][msg.sender])
                      && (from == msg.sender
                          || old(_allowances[from][msg.sender])
                             == type(uint256).max))))))
```

erc20-transferfrom-change-state

Function `transferFrom` Has No Unexpected State Changes.

All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` may only modify the following state variables:

- The balance entry for the address in `dest`,
- The balance entry for the address in `from`,
- The allowance for the address in `msg.sender` for the address in `from`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, amount), p1 != from && p1 != to
   && (p2 != from || p3 != msg.sender))
    ==> <>(finished(contract.transferFrom(from, to, amount), return
     ==> (_totalSupply == old(_totalSupply) && _balances[p1] == old(_balances[p1])
         && _allowances[p2][p3] == old(_allowances[p2][p3])  ))))
```

erc20-transferfrom-fail-exceed-balance

Function `transferFrom` Fails if the Requested Amount Exceeds the Available Balance.

Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the balance of address `from` must fail.

Specification:

```
[](started(contract.transferFrom(from, to, value), value > _balances[from]
   && _balances[from] >= 0 && _balances[from] <= type(uint256).max)
       ==> <>(reverted(contract.transferFrom)
              || finished(contract.transferFrom, !return)))
```

erc20-transferfrom-fail-exceed-allowance

Function `transferFrom` Fails if the Requested Amount Exceeds the Available Allowance.

Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the allowance of address `msg.sender` must fail.

Specification:

```
  [](started(contract.transferFrom(from, to, value), value > _allowances[from]
[msg.sender]
    && _allowances[from][msg.sender] >= 0 && value <= type(uint256).max)
    ==> <>(reverted(contract.transferFrom)
          || finished(contract.transferFrom(from, to, value), !return)
          || finished(contract.transferFrom(from, to, value), return
             && (msg.sender == from
                 || _allowances[from][msg.sender] == type(uint256).max))))
```

erc20-transferfrom-fail-recipient-overflow

Function `transferFrom` Prevents Overflows in the Recipient's Balance.

Any call of `transferFrom(from, dest, amount)` with a value in `amount` whose transfer would cause an overflow of the balance of address `dest` must fail.

Specification:

```
[](started(contract.transferFrom(from, to, value), from != to
   && _balances[to] + value > type(uint256).max && value <= type(uint256).max
   && _balances[to] >= 0 && _balances[to] <= type(uint256).max)
      ==> <>(reverted(contract.transferFrom)
          || finished(contract.transferFrom(from, to, value), !return)
          || finished(contract.transferFrom(from, to, value), _balances[to]
              > old(_balances[to]) + value - type(uint256).max - 1)))
```

erc20-transferfrom-false

If Function `transferFrom` Returns `false`, the Contract's State Has Not Been Changed.

If `transferFrom` returns `false` to signal a failure, it must undo all incurred state changes before returning to the caller.

Specification:

```
[](willSucceed(contract.transfer(to, value))
   ==> <>(finished(contract.transfer(to, value), !return
   ==> (_balances == old(_balances) && _totalSupply == old(_totalSupply)
                                    && _allowances == old(_allowances)  ))))
```

erc20-transferfrom-never-return-false

Function `transferFrom` Never Returns `false`.

The `transferFrom` function must never return `false`.

Specification:

```
[](!(finished(contract.transferFrom, !return)))
```

Properties related to function `totalSupply`

erc20-totalsupply-succeed-always

Function `totalSupply` Always Succeeds.

The function `totalSupply` must always succeeds, assuming that its execution does not run out of gas.

Specification:

```
[](started(contract.totalSupply) ==> <>(finished(contract.totalSupply)))
```

erc20-totalsupply-correct-value

Function `totalSupply` Returns the Value of the Corresponding State Variable.

The `totalSupply` function must return the value that is held in the corresponding state variable of contract contract.

Specification:

```
[](willSucceed(contract.totalSupply)
   ==> <>(finished(contract.totalSupply, return == _totalSupply)))
```

erc20-totalsupply-change-state

Function `totalSupply` Does Not Change the Contract's State.

The `totalSupply` function in contract contract must not change any state variables.

Specification:

```
[](willSucceed(contract.totalSupply)
   ==> <>(finished(contract.totalSupply, _totalSupply == old(_totalSupply)
          && _balances == old(_balances) && _allowances == old(_allowances)  )))
```

Properties related to function `balanceOf`

erc20-balanceof-succeed-always

Function `balanceOf` Always Succeeds.

Function `balanceOf` must always succeed if it does not run out of gas.

Specification:

```
[](started(contract.balanceOf) ==> <>(finished(contract.balanceOf)))
```

erc20-balanceof-correct-value

Function `balanceOf` Returns the Correct Value.

Invocations of `balanceOf(owner)` must return the value that is held in the contract's balance mapping for address `owner`.

Specification:

```
[](willSucceed(contract.balanceOf)
   ==> <>(finished(contract.balanceOf(owner), return == _balances[owner])))
```

erc20-balanceof-change-state

Function `balanceOf` Does Not Change the Contract's State.

Function `balanceOf` must not change any of the contract's state variables.

Specification:

```
[](willSucceed(contract.balanceOf)
   ==> <>(finished(contract.balanceOf(owner), _totalSupply == old(_totalSupply)
          && _balances == old(_balances)
          && _allowances == old(_allowances)  )))
```

Properties related to function `allowance`

erc20-allowance-succeed-always

Function `allowance` Always Succeeds.

Function `allowance` must always succeed, assuming that its execution does not run out of gas.

Specification:

```
[](started(contract.allowance) ==> <>(finished(contract.allowance)))
```

erc20-allowance-correct-value

Function `allowance` Returns Correct Value.

Invocations of `allowance(owner, spender)` must return the allowance that address `spender` has over tokens held by address `owner`.

Specification:

```
[](willSucceed(contract.allowance(owner, spender))
   ==> <>(finished(contract.allowance(owner, spender),
           return == _allowances[owner][spender])))
```

erc20-allowance-change-state

Function `allowance` Does Not Change the Contract's State.

Function `allowance` must not change any of the contract's state variables.

Specification:

```
[](willSucceed(contract.allowance(owner, spender))
   ==> <>(finished(contract.allowance(owner, spender),
   _totalSupply == old(_totalSupply) && _balances == old(_balances)
               && _allowances == old(_allowances)  )))
```

Properties related to function `approve`

erc20-approve-revert-zero

Function `approve` Prevents Giving Approvals For the Zero Address.

All calls of the form `approve(spender, amount)` must fail if the address in `spender` is the zero address.

Specification:

```
[](started(contract.approve(spender, value), spender == address(0))
   ==> <>(reverted(contract.approve)
          || finished(contract.approve(spender, value), !return)))
```

erc20-approve-succeed-normal

Function `approve` Succeeds for Admissible Inputs.

All calls of the form `approve(spender, amount)` must succeed, if

- the address in `spender` is not the zero address and
- the execution does not run out of gas.

Specification:

```
[](started(contract.approve(spender, value), spender != address(0))
   ==> <>(finished(contract.approve(spender, value), return)))
```

erc20-approve-correct-amount

Function `approve` Updates the Approval Mapping Correctly.

All non-reverting calls of the form `approve(spender, amount)` that return `true` must correctly update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount`.

Specification:

```
[](willSucceed(contract.approve(spender, value), spender != address(0)
   && value >= 0 && value <= type(uint256).max)
       ==> <>(finished(contract.approve(spender, value), return
              ==> _allowances[msg.sender][spender] == value)))
```

erc20-approve-change-state

Function `approve` Has No Unexpected State Changes.

All calls of the form `approve(spender, amount)` must only update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount` and incur no other state changes.

Specification:

```
[](willSucceed(contract.approve(spender, value), spender != address(0)
   && (p1 != msg.sender || p2 != spender))
       ==> <>(finished(contract.approve(spender, value), return
              ==> _totalSupply == old(_totalSupply) && _balances == old(_balances)
                  && _allowances[p1][p2] == old(_allowances[p1][p2])  )))
```

erc20-approve-false

If Function `approve` Returns `false`, the Contract's State Has Not Been Changed.

If function `approve` returns `false` to signal a failure, it must undo all state changes that it incurred before returning to the caller.

Specification:

```
[](willSucceed(contract.approve(spender, value))
   ==> <>(finished(contract.approve(spender, value), !return
          ==> (_balances == old(_balances) && _totalSupply == old(_totalSupply)
               && _allowances == old(_allowances)  ))))
```

erc20-approve-never-return-false

Function `approve` Never Returns `false`.

The function `approve` must never returns `false`.

Specification:

```
 [](!(finished(contract.approve, !return)))
```

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | Securing the Web3 World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.