



**PERCENT FINANCE SMART CONTRACT AUDIT
FOR DAO Percent Finance**

29.09.2020

Made in Germany by Chainsulting.de



Smart Contract Audit

1. Disclaimer.....	3
2. About the Project and Company.....	4
2.1 Project Overview.....	5
3. Vulnerability & Risk Level	6
4. Auditing Strategy and Techniques Applied.....	7
4.1 Methodology	7
4.2 Used Code from other Frameworks (Changes).....	8
4.3 Tested Contract Files	13
4.4 Contract Specifications (ERC20 Token)	13
5. Summary of Smart Contract (ChainlinkPriceOracleProxy).....	14
5.1 Visualized Dependencies	14
5.2 Functions.....	15
5.3 Modifiers.....	18
5.4 States.....	18
6. Test Suite Results.....	20
6.1 Mythril Classic & MYTHX Security Audit.....	20
6.2. SWC Attacks	21
7. Executive Summary.....	25
8. Deployed Smart Contract	26
9. Compound Finance Audit (By OpenZeppelin / 23.09.2019)	27



1. Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of Percent Finance. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Major Versions / Date	Description
0.1 (25.09.2020)	Layout
0.5 (26.09.2020)	Automated Security Testing Manual Security Testing
1.0 (27.09.2020)	Summary and Recommendation
1.1 (28.09.2020)	Adding of MythX and SWC
2.0 (29.09.2020)	Final document

2. About the Project and Company

Company address: N/A (DAO)

Voting Governance: <https://snapshot.page/#/percent>

Team: N/A

Website: <https://percent.finance>

GitHub: <https://github.com/percent-finance/>

Twitter: <https://twitter.com/PercentFinance>

Discord: <https://discord.com/invite/3sxWhH3>

Medium: <https://medium.com/percent-finance>

2.1 Project Overview

The Percent protocol is a decentralized money market that offers users access to permissionless lending and borrowing. In essence, they are a fork of Compound that is powered by Chainlink oracles. Their aim is to create a truly decentralized lending/borrowing platform with fair distribution, fair governance, and unmanipulated data oracles.

In order to accomplish these goals they have implemented: Fair token distribution with 100% token allocation to liquidity providers with no pre-mining or reserves for the team, no unannounced launch, Chainlink price oracle integration for providing a wide range of accurate and decentralized price feeds, Governance token PCT with a quadratic voting mechanism to incentivize the number of participants involved in platform governance.

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 – 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	4 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	2 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0 – 1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk

4. Auditing Strategy and Techniques Applied

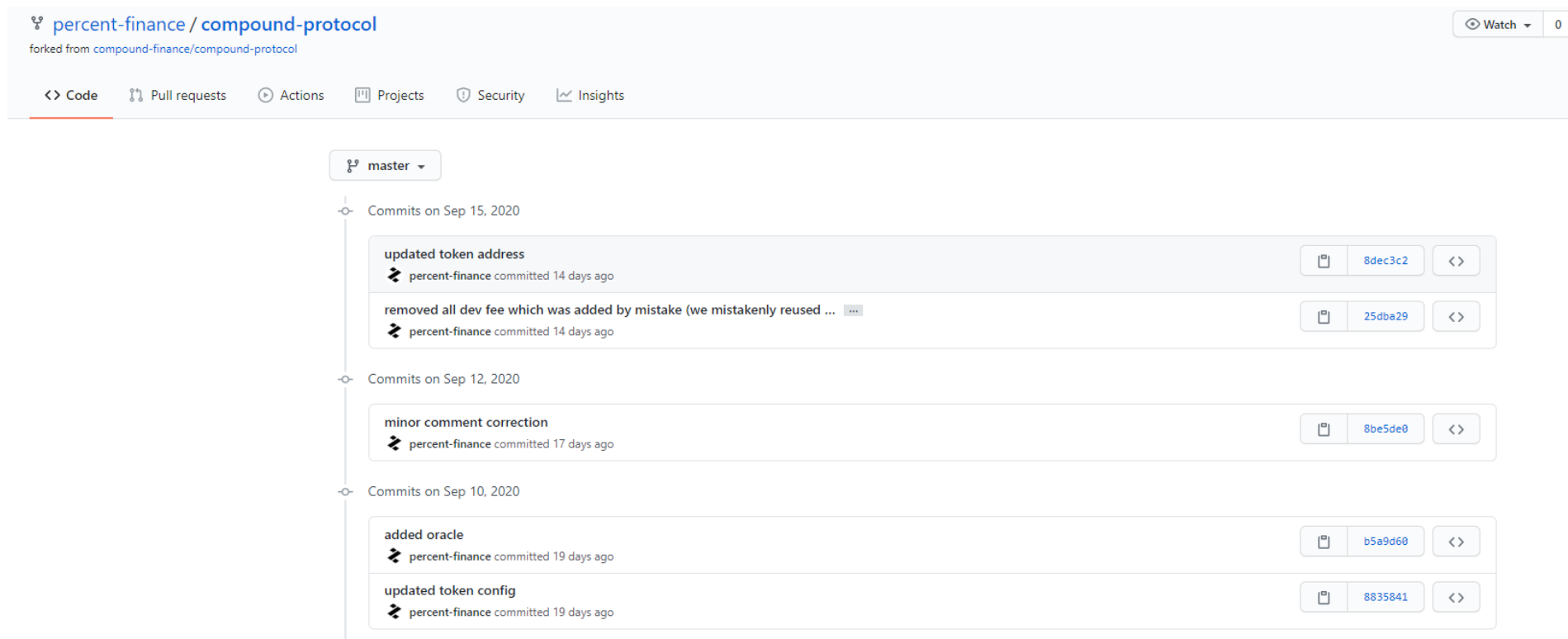
Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert pentesters and smart contract developers, documenting any issues as there were discovered.

4.1 Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
 - i. Review of the specifications, sources, and instructions provided to Chainsulting to make sure we understand the size, scope, and functionality of the smart contract.
 - ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
 - iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Chainsulting describe.
2. Testing and automated analysis that includes the following:
 - i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
 - ii. Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

4.2 Used Code from other Frameworks (Changes)



The screenshot displays the GitHub interface for the repository `percent-finance / compound-protocol`, which is a fork of `compound-finance/compound-protocol`. The repository is currently on the `master` branch. The commit history is filtered by date, showing commits from September 10, 2020, to September 15, 2020.

Commits on Sep 15, 2020:

- `updated token address` (commit `8dec3c2`) by percent-finance, committed 14 days ago.
- `removed all dev fee which was added by mistake (we mistakenly reused ...` (commit `25dba29`) by percent-finance, committed 14 days ago.

Commits on Sep 12, 2020:

- `minor comment correction` (commit `8be5de0`) by percent-finance, committed 17 days ago.

Commits on Sep 10, 2020:

- `added oracle` (commit `b5a9d60`) by percent-finance, committed 19 days ago.
- `updated token config` (commit `8835841`) by percent-finance, committed 19 days ago.

Fork:

<https://github.com/percent-finance/compound-protocol/commits/master>

Original Source:

<https://github.com/compound-finance/compound-protocol/commits/master>

The following changes has been made by the percent finance development team:

4.2.1 Update Token Config

Link	Commit ID	Summary
https://github.com/percent-finance/compound-protocol/commit/8835841266c3fb509e6553716808cfb373762af6	8835841266c3fb509e6553716808cfb373762af6	Updated Token Config

Code:

```
contracts/Governance/Comp.sol
@@ -3,16 +3,16 @@ pragma experimental ABIEncoderV2;

3 3
4 4 contract Comp {
5 5     /// @notice EIP-20 token name for this token
6 - string public constant name = "Compound";
6 + string public constant name = "Percent";

7 7
8 8     /// @notice EIP-20 token symbol for this token
9 - string public constant symbol = "COMP";
9 + string public constant symbol = "PCT";

10 10
11 11     /// @notice EIP-20 token decimals for this token
12 12     uint8 public constant decimals = 18;
13 13
14 14     /// @notice Total number of tokens in circulation
15 - uint public constant totalSupply = 10000000e18; // 10 million Comp
15 + uint public constant totalSupply = 20000000e18; // 10 million PCT

16 16
17 17     /// @notice Allowance amounts on behalf of others
18 18     mapping (address => mapping (address => uint96)) internal allowances;
```

4.2.2 Chainlink oracle

Link	Commit ID	Summary
https://github.com/percent-finance/compound-protocol/commit/b5a9d6025f54c4f9b85399817c0acf7df9be0e59	b5a9d6025f54c4f9b85399817c0acf7df9be0e59	Added Chainlink price oracle

Check Percent Finance Chainlink oracle integration in our audit

4.2.3 Minor comment correction

Link	Commit ID	Summary
https://github.com/percent-finance/compound-protocol/commit/8be5de032b010935d6daebf93804f2d2d69961f1	8be5de032b010935d6daebf93804f2d2d69961f1	Minor comment correction

Code:

```
contracts/Governance/Comp.sol
@@ -12,7 +12,7 @@ contract Comp {
12 12     uint8 public constant decimals = 18;
13 13
14 14     /// @notice Total number of tokens in circulation
15 -   uint public constant totalSupply = 20000000e18; // 10 million PCT
15 +   uint public constant totalSupply = 20000000e18; // 20 million PCT
16 16
17 17     /// @notice Allowance amounts on behalf of others
18 18     mapping (address => mapping (address => uint96)) internal allowances;
```

4.2.4 Removed dev fees

Link	Commit ID	Summary
https://github.com/percent-finance/compound-protocol/commit/25dba29888c49d202e67f2924bb45bbb79124d9a	25dba29888c49d202e67f2924bb45bbb79124d9a	removed all dev fee which was added by mistake (we mistakenly reused ... the code from the previous iteration)

Code:

```

contracts/PctPool.sol
@@ -603,11 +603,9 @@ contract PctPool is LPTokenWrapper, IRewardDistributionRecipient {
    uint256 public rewardPerTokenStored;
    mapping(address => uint256) public userRewardPerTokenPaid;
    mapping(address => uint256) public rewards;
-   address public devAddr;
    constructor(address pctAddress, address stakeTokenAddress) LPTokenWrapper(stakeTokenAddress) public {
        rewardDistribution = msg.sender;
-   devAddr = msg.sender;
        pct = IERC20(pctAddress);
    }
}

@@ -674,10 +672,8 @@ contract PctPool is LPTokenWrapper, IRewardDistributionRecipient {
    uint256 reward = earned(msg.sender);
    if (reward > 0) {
        rewards[msg.sender] = 0;
-   uint256 devFee = reward.mul(8).div(100);
-   pct.safeTransfer(devAddr, devFee);
-   pct.safeTransfer(msg.sender, reward.sub(devFee));
-   emit RewardPaid(msg.sender, reward.sub(devFee));
+   pct.safeTransfer(msg.sender, reward);
+   emit RewardPaid(msg.sender, reward);
    }
}

```

4.2.5 Updated Token Address

Link	Commit ID	Summary
https://github.com/percent-finance/compound-protocol/commit/8dec3c293b3a9087bd67d0f6b7082df421978d61	8dec3c293b3a9087bd67d0f6b7082df421978d61	Updated token address

Code:

```
contracts/Comptroller.sol
@@ -1376,6 +1376,6 @@ contract Comptroller is ComptrollerV3Storage, ComptrollerInterface, ComptrollerE
1376 1376      * @return The address of COMP
1377 1377      */
1378 1378      function getCompAddress() public view returns (address) {
1379 -      return 0xc00e94Cb662C3520282E6f5717214004A7f26888;
1379 +      return 0xbc16da9df0A22f01A16BC0620a27e7D6d6488550;
1380 1380      }
1381 1381      }
```

4.3 Tested Contract Files

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review

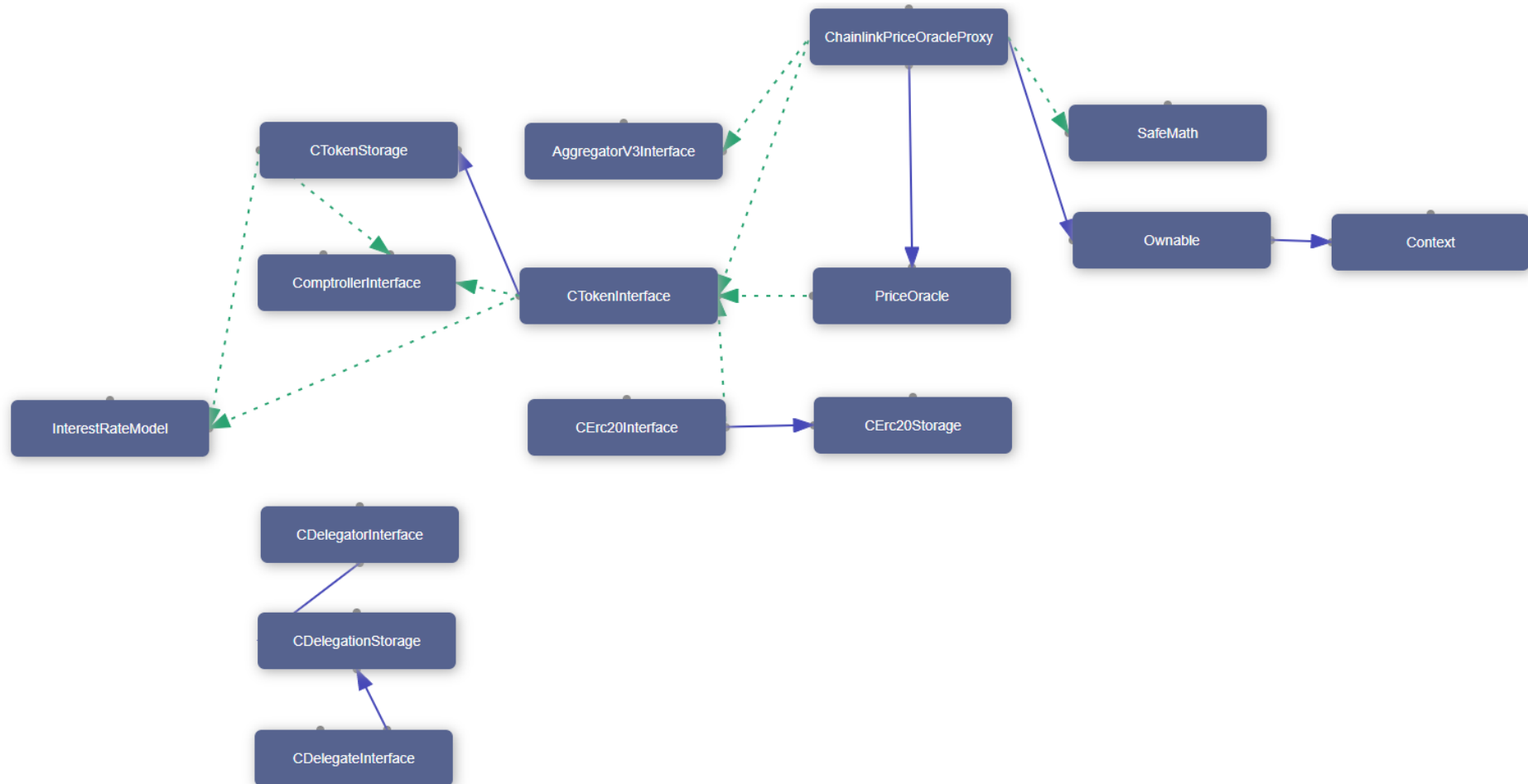
File	Fingerprint (SHA256)	Source
ChainlinkPriceOracleProxy.sol	407027b5ff7fe798b7091b7edc90d77d0f41dff0897642ec5a406af18d6e7448	https://raw.githubusercontent.com/chainsulting/Smart-Contract-Security-Audits/master/Percent%20Finance/percent-finance-protocol/contracts/ChainlinkPriceOracleProxy.sol
percentfinance.sol	6b652ce95582f0e5fc619653d11da237096026ea5ea785f330d42c095ad47a16	https://raw.githubusercontent.com/chainsulting/Smart-Contract-Security-Audits/master/Percent%20Finance/erc20-token/percentfinance.sol

4.4 Contract Specifications (ERC20 Token)

Language	Solidity
Token Standard	ERC20
Compiler Version	0.5.16
Upgradeable	No
Burn Function	No
Delegate	Yes
Mint Function	No (Fixed total supply)
Lock Mechanism	No
Vesting Function	No
Ticker Symbol	PCT
Total Supply	20 000 000
Decimals	18

5. Summary of Smart Contract (ChainlinkPriceOracleProxy)

5.1 Visualized Dependencies



5.2 Functions

contract	func	visibility	modifiers	stateMutability
CTokenInterface	transfer	external		
CTokenInterface	transferFrom	external		
CTokenInterface	approve	external		
CTokenInterface	allowance	external		view
CTokenInterface	balanceOf	external		view
CTokenInterface	balanceOfUnderlying	external		
CTokenInterface	getAccountSnapshot	external		view
CTokenInterface	borrowRatePerBlock	external		view
CTokenInterface	supplyRatePerBlock	external		view
CTokenInterface	totalBorrowsCurrent	external		
CTokenInterface	borrowBalanceCurrent	external		
CTokenInterface	borrowBalanceStored	public		view
CTokenInterface	exchangeRateCurrent	public		
CTokenInterface	exchangeRateStored	public		view
CTokenInterface	getCash	external		view
CTokenInterface	accrueInterest	public		
CTokenInterface	seize	external		
CTokenInterface	_setPendingAdmin	external		
CTokenInterface	_acceptAdmin	external		
CTokenInterface	_setComptroller	public		
CTokenInterface	_setReserveFactor	external		

CTokenInterface	_reduceReserves	external		
CTokenInterface	_setInterestRateModel	public		
CErc20Interface	mint	external		
CErc20Interface	redeem	external		
CErc20Interface	redeemUnderlying	external		
CErc20Interface	borrow	external		
CErc20Interface	repayBorrow	external		
CErc20Interface	repayBorrowBehalf	external		
CErc20Interface	liquidateBorrow	external		
CErc20Interface	_addReserves	external		
CDelegatorInterface	_setImplementation	public		
CDelegateInterface	_becomeImplementation	public		
CDelegateInterface	_resignImplementation	public		
SafeMath	add	internal		pure
SafeMath	add	internal		pure
SafeMath	sub	internal		pure
SafeMath	sub	internal		pure
SafeMath	mul	internal		pure
SafeMath	mul	internal		pure
SafeMath	div	internal		pure
SafeMath	div	internal		pure
SafeMath	mod	internal		pure
SafeMath	mod	internal		pure
ComptrollerInterface	enterMarkets	external		
ComptrollerInterface	exitMarket	external		

ComptrollerInterface	mintAllowed	external		
ComptrollerInterface	mintVerify	external		
ComptrollerInterface	redeemAllowed	external		
ComptrollerInterface	redeemVerify	external		
ComptrollerInterface	borrowAllowed	external		
ComptrollerInterface	borrowVerify	external		
ComptrollerInterface	repayBorrowAllowed	external		
ComptrollerInterface	repayBorrowVerify	external		
ComptrollerInterface	liquidateBorrowAllowed	external		
ComptrollerInterface	liquidateBorrowVerify	external		
ComptrollerInterface	seizeAllowed	external		
ComptrollerInterface	seizeVerify	external		
ComptrollerInterface	transferAllowed	external		
ComptrollerInterface	transferVerify	external		
ComptrollerInterface	liquidateCalculateSeizeTokens	external		view
InterestRateModel	getBorrowRate	external		view
InterestRateModel	getSupplyRate	external		view
Context	_msgSender	internal		view
Context	_msgData	internal		view
Ownable	"constructor"	internal		
Ownable	owner	public		view
Ownable	renounceOwnership	public	onlyOwner	
Ownable	transferOwnership	public	onlyOwner	
AggregatorV3Interface	decimals	external		view
AggregatorV3Interface	description	external		view

AggregatorV3Interface	version	external		view
AggregatorV3Interface	getRoundData	external		view
AggregatorV3Interface	latestRoundData	external		view
PriceOracle	getUnderlyingPrice	external		view
ChainlinkPriceOracleProxy	"constructor"	public		
ChainlinkPriceOracleProxy	getUnderlyingPrice	public		view
ChainlinkPriceOracleProxy	setEthUsdChainlinkAggregatorAddress	external	onlyOwner	
ChainlinkPriceOracleProxy	setTokenConfigs	external	onlyOwner	

5.3 Modifiers

contract	modifier
Ownable	onlyOwner

5.4 States

contract	state	type	visibility	isConst
CTokenStorage	_notEntered	bool	internal	false
CTokenStorage	name	string	public	false
CTokenStorage	symbol	string	public	false
CTokenStorage	decimals	uint8	public	false
CTokenStorage	borrowRateMaxMantissa	uint	internal	true
CTokenStorage	reserveFactorMaxMantissa	uint	internal	true
CTokenStorage	admin	address	public	false
CTokenStorage	pendingAdmin	address	public	false
CTokenStorage	comptroller	ComptrollerInterface	public	false

CTokenStorage	interestRateModel	InterestRateModel	public	false
CTokenStorage	initialExchangeRateMantissa	uint	internal	false
CTokenStorage	reserveFactorMantissa	uint	public	false
CTokenStorage	accrualBlockNumber	uint	public	false
CTokenStorage	borrowIndex	uint	public	false
CTokenStorage	totalBorrows	uint	public	false
CTokenStorage	totalReserves	uint	public	false
CTokenStorage	totalSupply	uint	public	false
CTokenStorage	accountTokens	mapping	internal	false
CTokenStorage	transferAllowances	mapping	internal	false
CTokenStorage	accountBorrows	mapping	internal	false
CTokenInterface	isCToken	bool	public	true
CErc20Storage	underlying	address	public	false
CDelegationStorage	implementation	address	public	false
ComptrollerInterface	isComptroller	bool	public	true
InterestRateModel	isInterestRateModel	bool	public	true
Ownable	_owner	address	private	false
PriceOracle	isPriceOracle	bool	public	true
ChainlinkPriceOracleProxy	isPriceOracle	bool	public	true
ChainlinkPriceOracleProxy	ethUsdChainlinkAggregatorAddress	address	public	false
ChainlinkPriceOracleProxy	tokenConfig	mapping	public	false

6. Test Suite Results

The Chainlink Oracle integration is part of the Percent Finance Smart Contract fork of compound finance and this one was audited. All the functions and state variables are well commented using the natspec documentation for the functions which is good to understand quickly how everything is supposed to work.

6.1 Mythril Classic & MYTHX Security Audit

Mythril Classic is an open-source security analysis tool for Ethereum smart contracts. It uses concolic analysis, taint analysis and control flow checking to detect a variety of security vulnerabilities.

Findings: None

6.2. SWC Attacks

ID	Title	Relationships	Test Result
SWC-131	Presence of unused variables	CWE-1164: Irrelevant Code	✓
SWC-130	Right-To-Left-Override control character (U+202E)	CWE-451: User Interface (UI) Misrepresentation of Critical Information	✓
SWC-129	Typographical Error	CWE-480: Use of Incorrect Operator	✓
SWC-128	DoS With Block Gas Limit	CWE-400: Uncontrolled Resource Consumption	✓
SWC-127	Arbitrary Jump with Function Type Variable	CWE-695: Use of Low-Level Functionality	✓
SWC-125	Incorrect Inheritance Order	CWE-696: Incorrect Behavior Order	✓
SWC-124	Write to Arbitrary Storage Location	CWE-123: Write-what-where Condition	✓
SWC-123	Requirement Violation	CWE-573: Improper Following of Specification by Caller	✓

ID	Title	Relationships	Test Result
SWC-122	Lack of Proper Signature Verification	CWE-345: Insufficient Verification of Data Authenticity	✓
SWC-121	Missing Protection against Signature Replay Attacks	CWE-347: Improper Verification of Cryptographic Signature	✓
SWC-120	Weak Sources of Randomness from Chain Attributes	CWE-330: Use of Insufficiently Random Values	✓
SWC-119	Shadowing State Variables	CWE-710: Improper Adherence to Coding Standards	✓
SWC-118	Incorrect Constructor Name	CWE-665: Improper Initialization	✓
SWC-117	Signature Malleability	CWE-347: Improper Verification of Cryptographic Signature	✓
SWC-116	Timestamp Dependence	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✓
SWC-115	Authorization through tx.origin	CWE-477: Use of Obsolete Function	✓
SWC-114	Transaction Order Dependence	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	✓

ID	Title	Relationships	Test Result
SWC-113	DoS with Failed Call	CWE-703: Improper Check or Handling of Exceptional Conditions	✓
SWC-112	Delegatecall to Untrusted Callee	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✓
SWC-111	Use of Deprecated Solidity Functions	CWE-477: Use of Obsolete Function	✓
SWC-110	Assert Violation	CWE-670: Always-Incorrect Control Flow Implementation	✓
SWC-109	Uninitialized Storage Pointer	CWE-824: Access of Uninitialized Pointer	✓
SWC-108	State Variable Default Visibility	CWE-710: Improper Adherence to Coding Standards	✓
SWC-107	Reentrancy	CWE-841: Improper Enforcement of Behavioral Workflow	✓
SWC-106	Unprotected SELFDESTRUCT Instruction	CWE-284: Improper Access Control	✓
SWC-105	Unprotected Ether Withdrawal	CWE-284: Improper Access Control	✓
SWC-104	Unchecked Call Return Value	CWE-252: Unchecked Return Value	✓

ID	Title	Relationships	Test Result
SWC-103	Floating Pragma	CWE-664: Improper Control of a Resource Through its Lifetime	✓
SWC-102	Outdated Compiler Version	CWE-937: Using Components with Known Vulnerabilities	✓
SWC-101	Integer Overflow and Underflow	CWE-682: Incorrect Calculation	✓
SWC-100	Function Default Visibility	CWE-710: Improper Adherence to Coding Standards	✓

Sources:

<https://smartcontractsecurity.github.io/SWC-registry>

<https://dasp.co>

<https://github.com/chainsulting/Smart-Contract-Security-Audits>

https://consensys.github.io/smart-contract-best-practices/known_attacks

7. Executive Summary

A majority of the code was standard and copied from widely-used and reviewed contracts and as a result, a lot of the code was reviewed before. It correctly implemented widely-used and reviewed contracts for safe mathematical operations. The audit identified no major security vulnerabilities, at the moment of audit. We noted that a majority of the functions were self-explanatory, and standard documentation tags (such as `@dev`, `@param`, and `@returns`) were included. No critical issues were found after the manual and automated security testing. (Testing included PCT Token and Chainlink integration)

8. Deployed Smart Contract

PCT (Percent Token)

<https://etherscan.io/address/0xbc16da9df0a22f01a16bc0620a27e7d6d6488550>

Kovan Testnet ChainlinkPriceOracleProxy

<https://kovan.etherscan.io/address/0x2722e306825b09be6e6F3AC50cAAeaAf9386992C#code>

9. Compound Finance Audit (By OpenZeppelin / 23.09.2019)

[Compound Finance](#) is a protocol, currently deployed on the Ethereum network, for automatic, permissionless, and trust-minimized loans of Ether and various ERC20 tokens. It is one of the most widely used decentralized finance systems in the ecosystem and helps demonstrate the power of the technology.

The team asked us to review and audit a subset of the smart contracts. We reviewed the code and now publish our results.

The audited commit is f385d71983ae5c5799faae9b2dfea43e5cf75262 and the files included in the scope were:

[CarefulMath](#), [CErc20](#), [CEther](#), [Comptroller](#), [ComptrollerInterface](#), [ComptrollerV1Storage](#), [UnitrollerAdminStorage](#), [CToken](#), [EIP20Interface](#), [EIP20NonStandardInterface](#), [ComptrollerErrorReporter](#), [TokenErrorReporter](#), [Exponential](#), [InterestRateModel](#), [Maximillion](#), [ReentrancyGuard](#), [Unitroller](#) and [WhitePaperInterestRateModel](#).

The Price Oracle system is intended to be replaced shortly and was therefore not reviewed. During this audit we assumed that the administrator and price feeds are available, honest and not compromised.

Here we present our findings.

Critical Severity

None.

High Severity

Interest-Free Loans

The [CToken contract](#) calculates the borrow balance of an account in the function [borrowBalanceStoredInternal](#).

The balance is the principal scaled by the ratio of the borrow indices (which track the accumulative effect of per-block interest changes):

```
balance = principal * (current borrow index) / (original borrow index)
```

However, when the principal and ratio of borrow indices are both small the result can equal the principal, due to automatic truncation of division within solidity.

This means that a loan could accrue no actual interest, yet still be factored into calculations of `totalReserves`, `totalBorrows`, and `exchangeRates`. In the case of many small loans being taken out, the associated interest calculated for that market may not match the amount actually received when users pay off those loans.

In brief: it is possible for users to take out small, short-term, interest-free loans. Optionally, they can then resupply the borrowed assets back into Compound to receive interest.

An example attack works as follows:

1. The attacker takes out several interest-free loans. By the nature of this vulnerability, these loans must be small. However, the attacker can take out arbitrarily many of them. It is most effective if the attacker borrows an asset for which the value of this truncation error is greatest (currently, wBTC).
2. The attacker consolidates the small interest-free loans by sending all the borrowed assets to a single Ethereum account.
3. (Optional) The attacker swaps the borrowed assets for an equal value of the highest-interest-rate asset on Compound (currently DAI).
4. The attacker deposits the asset into Compound.
5. (Optional) The attacker can then repeat the process — leveraging up — if they so desire.
6. The attacker unwinds the trade and pays off all the small loans before the loans start “registering” interest (that is, before the interest owed is no longer “truncated away”). The result is that the attacker collects the supply-interest but does not have to pay the borrow-interest. Note that this is mathematically equivalent to the attacker simply stealing cash.
7. Repeat.

There are a few practical considerations that make this attack non-trivial to pull off on the live network. These difficulties have to do with things unrelated to the Compound protocol: gas costs, slippage, and the unpredictable behavior of other Compound users. All of these can be overcome by certain classes of attackers, and we address them below.

In practice, steps 1, 2, and 6 of this process would be gas-intensive. So much so that the attack would not be profitable if the attacker has to pay a normal gas price. However, if the attacker is a miner then their gas price is zero, and so this attack costs them nothing in gas. (An exception is if the blocks are full, in which case the attacking miner must pay the opportunity cost of not filling the block with normal, paying transactions). Note that the miner does not need much mining power to do this. The ability to mine a couple of blocks per week is technically sufficient — though the more mining power they have, the more money they can borrow with no interest, and the more likely they are to be able to unwind the trade in time.

Additionally, if the attacker wants to get the most out of the attack, they'll likely want to perform the optional step 3. This requires that they exchange one asset for another on the open market. Here they may incur fees and/or slippage as they sell one asset for another. If fees and/or slippage are greater than the profit they make on their short-term, interest-free loan, then the attack is not profitable. This limits the attackers to those that can make trades very cheaply (or to those who are willing to forgo step 3 and accept a smaller rate of return).

Finally, the attacker may be foiled if other users of Compound borrow more of the same asset the attacker did — thus driving the borrow interest rate up and causing the attacker's owed interest to unexpectedly rise above the level at which it would be "truncated away". An attacker has two ways to mitigate this risk. First, they can borrow smaller amounts to give themselves some "cushion" just in case the interest rate on their borrow rises. Second, they can choose to only go long on assets where they would be okay paying the interest anyway. That way, if the attack fails, they end up in the same position as anyone else going long on that asset — and if the attack succeeds, then they get the added bonus of not having to pay interest for the assets they used for leverage.

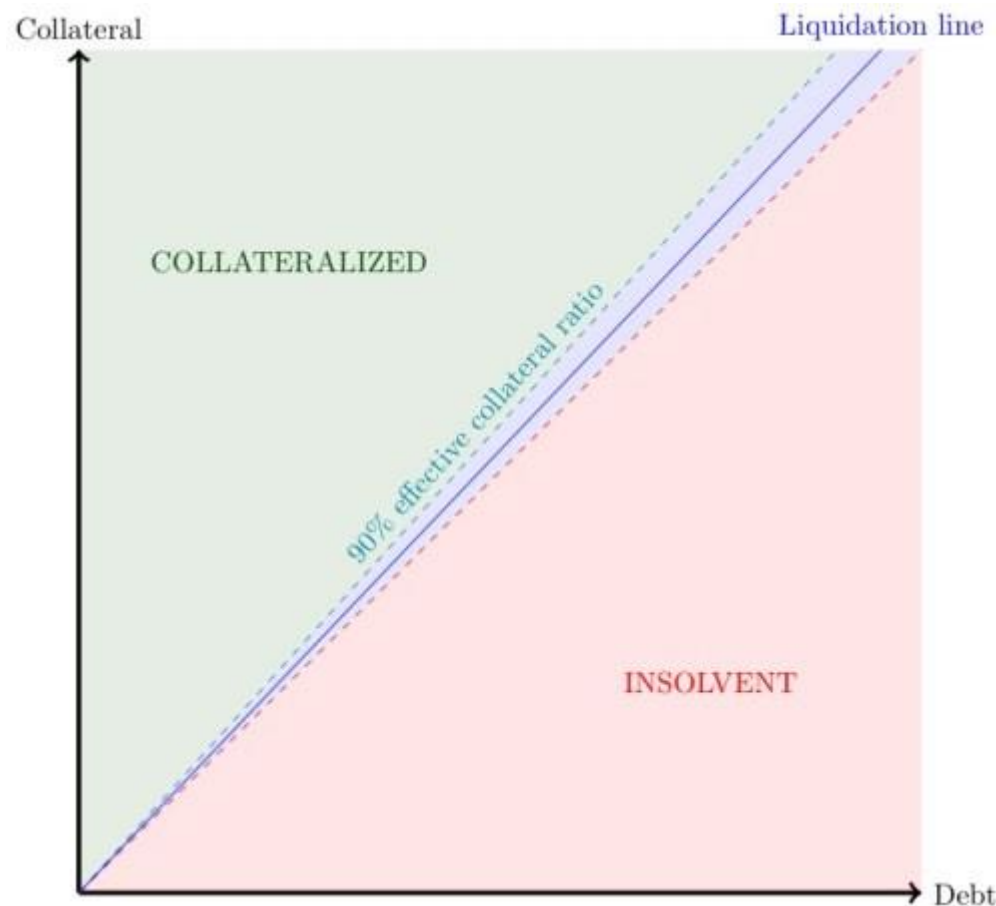
In short, there exists unavoidable error when computing interest. However, there does exist a choice over who gets the benefit of that error. The error should always be handled in a way that benefits the cToken contract, not the borrower. The small rounding error in favor of the borrower (and at the expense of the cToken contract) can be scaled up to eventually steal arbitrarily large sums of money from the cToken contract.

Consider adjusting the code such that the calculated borrow balance is always rounded *up* instead of being truncated.

Counterproductive Incentives

The protocol includes a mechanism to incentivize arbitrageurs to repay loans that are under-collateralized. This should increase the borrower's liquidity, reducing the risk of insolvency. After all, that is the point of the mechanism. However, there is a threshold where the incentives reverse and the liquidation mechanism actually pushes borrowers towards insolvency.

Consider a borrower with cTokens in various markets worth a total of C , and total debt worth D . They are located at point (D, C) in the following diagram:



After accounting for the distribution of cTokens and their corresponding collateral factors (the fraction of cTokens that can be used as collateral in each market), we can assign them an effective collateral ratio (ECR). The maximum possible value is 90%, which is labeled on the diagram. It corresponds to the minimum possible slope for remaining in the collateralized region, which is $1 / 90\% = 1.11$. With these values, we can describe the important regions:

- Users in the green region above the ECR line are fully collateralized. They have enough collateral that even after reducing it by the ECR, it is still larger than their debt.

- Users in the red region are insolvent. Their debt is larger than their collateral and they no longer have any incentive to repay it. Users in this region can be a major problem for the market because they continue accruing interest while removing liquidity from the market, ensuring at least some cToken holders will be unable to redeem their assets.
- Users in the blue region are under-collateralized. They still have more assets than debt in the system, but the protocol deems them to be at risk of becoming insolvent.

When a borrower is under-collateralized, anyone can repay some of their debt in exchange for collateral at better-than-market rates. This mechanism is parameterized by two global values that are set by the administrator:

- the close factor: the maximum fraction of the original loan that can be liquidated
- the liquidation incentive L : how much collateral do they receive (as a multiplier of the amount paid by the liquidator)

The Liquidation line on the diagram has slope L and passes through the origin. When an arbitrageur repays x debt, they receive $x \cdot L$ collateral from the borrower. In other words, they force the borrower to the bottom-left of the diagram on a trajectory parallel to the Liquidation line. The crucial observation is that this process can never cause a borrower to cross the liquidation line. Borrowers above the line are liquidated until they are fully collateralized, but borrowers below the line are actually pushed further towards insolvency by the liquidation process.

Ideally, they would already be liquidated before they crossed the threshold, but this would depend on various factors outside of the protocol, such as the speed of price changes, the liveness of the oracle, the congestion of Ethereum and the responsiveness and liquidity of arbitrageurs.

The Compound system currently has a liquidation incentive of 1.05, **which means that when a borrower's debt rises to at least $1 / 1.05 = 95.2\%$ of the value of their cTokens, each liquidation will push them further towards insolvency**. The maximum amount that they can be liquidated in one transaction is bounded by the close factor. The current close factor in the Compound system is 0.5, **which means that when a borrower's debt rises to at least 97.6% of the value of their cTokens, it is possible for a liquidation to force them into insolvency**.

Consider using a dynamic liquidation incentive that lowers as borrowers approach insolvency to ensure liquidations always increase solvency. Alternatively, consider treating the “incentivized insolvency” threshold of 95.2% as equivalent to insolvency, and choose the collateral factors and liquidation incentive accordingly.

Medium Severity

When calculating interest in `CToken accrueInterest()`, simple interest is applied over the blocks since the last update. This will underestimate the amount of interest that would be calculated if it were compounded every block.

The code is designed to accrue interest as frequently as possible, but this requirement expands the responsibility of accruing interest into otherwise unrelated functions. Additionally, the size of the discrepancy between the computed and theoretical interest will depend on the volume of transactions being handled by the Compound protocol, which may change unpredictably.

To improve predictability and functional encapsulation, consider calculating interest with the compound interest formula, rather than simulating it through repeated transactions. Note that the additional gas requirements may be reduced using the [modexp precompile](#). Separately, consider measuring the time between calls to `accrueInterest()` using *seconds* rather than *blocks*. This will help keep the interest rate calculation robust against changes to the average blocktime.

Error Propagation

The codebase uses an error propagation mechanism that works as follows:

- A function that is supposed to return value X, instead returns a tuple (NO_ERROR, X)
- A function that would normally error out instead returns a tuple (ERROR_NAME, 0)
- Note that in both cases, the first return value is an error code (an enum type named Error)
- Whenever a function returns an error, the calling function has to check it and then bubbles it up (ie. the caller returns the error tuple, possibly adding context to the error message)
- At the top level, the error/context codes are emitted in an event and then the error code is returned. The transaction is successful.
- The UI will need to convert the error codes into an error message

This scheme has the benefit of providing useful failure messages when an operation fails. However, there are serious negative properties of this pattern which we discuss below.

- First, the error propagation scheme violates the “Fail Early and Loudly” principle. Errors are not trapped when they are found. Callers have to remember to check and handle the returned error codes. Failure to do so can have serious security consequences.
- The scheme also increases both the size and the complexity of the code base. We estimate that abandoning the current scheme in favor of the traditional require/assert/revert paradigm would reduce the size of the (Solidity) codebase by 40%-60%. This would, additionally, make the code itself easier to read and understand — which is an often overlooked but important property of secure code.

- The increased code complexity is not just a matter of cognitive overhead for those attempting to read/understand the code. It also results in very large increases in gas costs due to all the extra required opcodes. From the user's perspective, this is a strong argument against this pattern.
- Implementation of the scheme makes it incompatible with battle-tested libraries (like SafeMath) and requires the use of custom math functions that reproduce the same functionality except that they return errors instead of throwing on failure.
- The default value of an uninitialized `Error` enum is the `Error.NO_ERROR` value. This means that if a new `Error` variable is declared and returned by a function without having been set, the caller will assume that there was no error. That is, the default assumption by error handlers is that everything went according to plan. It would be safer to assume, by default, that things did *not* go according to plan unless verified otherwise. While this property is not inherent to this error-handling pattern, it is a feature of the particular implementation we audited.
- It does not cover every case, leading to inconsistencies within the code base. For example, the [CEther_mint function](#) reverts on failure, whereas the [CErc20_mint function](#) returns an error code.
- Finally, lack of a `revert` on failure is counter to what most Ethereum users have come to expect. For instance, a failed call to `CToken.transfer` via MetaMask will result in a `success` message from MetaMask even though the transfer failed. While this may not technically violate the ERC20 standard, it is counter-intuitive and increases the probability of user error.

With all of that said, it appears that all errors *are* properly handled in the code we audited. If considering a refactor of the contracts at some point in the future, we strongly recommend moving away from this error propagation pattern and instead using off-chain tools to evaluate failed transactions in order to display meaningful error messages to users.

Unknown Repayment Amount

To repay an ERC20 loan, a borrower can call [repayBorrow](#) or [repayBorrowBehalf](#) with a specified amount to repay.

However, they accrue interest in every block, which means if they specify the value of the loan at a particular block, their loan will be slightly higher in a future block when the transaction is confirmed and they will end up leaving part of the loan unpaid.

On the other hand, they could set the repay amount to `uint256(-1)`, which is used as a signal to mean “pay the whole loan”. This prevents the user from setting an upper bound on how much to pay. It is also particularly vulnerable in the case where an address is repaying on behalf of another address, since the borrower could front-run the transaction and borrow additional funds (up to the amount that the message sender has authorized the `CToken` contract to spend), which would also be repaid in the same transaction.

Consider treating the specified repayment amount as an upper bound, where the transaction repays the minimum of that value and the size of the loan.

Unsafe Assumptions About Average Time Between Blocks

The current implementation of the protocol uses *blocks* rather than *seconds* to measure time between interest accruals. This makes the implementation highly sensitive to changes in the average time between Ethereum blocks.

On [line 30 of WhitePaperInterestRateModel.sol](#) it is implicitly assumed that the time between blocks is 15 seconds. However, the average time between blocks can change dramatically.

For example, the average time between blocks may increase by significant factors due to the difficulty bomb or decrease by significant factors during the transition to Serenity.

The difference between the actual time between blocks and the assumed time between blocks causes proportional differences between the intended interest rates and the actual interest rates.

While it is possible for the admin to combat this by adjusting the interest rate model when the average time between blocks changes, such adjustments are manual and happen only after-the-fact. Errors in blocktime assumptions are cumulative, and fixing the model after-the-fact does not make users whole – it only prevents incorrect interest calculations moving forward (until the next change in blocktime).

Consider refactoring the implementation to use *seconds* rather than *blocks* to measure the time between accruals. While `block.timestamp` can be manipulated by miners within a narrow window, these errors are small and, importantly, are not cumulative. This would decouple the interest rate model from Ethereum's average blocktime.

Misleading NatSpec Comments

Since the purpose of the Ethereum Natural Specification (NatSpec) is to describe the code to the user, misleading statements should be considered a violation of the public API that may confuse or mislead users.

The `getBorrowRate` [interface](#) and [implementation](#) `@return` comments state that the rate is scaled by 10e18. In fact, it is only scaled by 1e18.

The CToken contract [borrowRateMaxMantissa](#) `comment` states that the maximum borrow rate per block is 0.0005% but it is actually 0.0005 (or 0.05%).

The comment on [line 7 of Unitroller.sol](#) is an incomplete thought/sentence.

The comment on [line 41 of ComptrollerStorage.sol](#) uses the word “discount” when it should use “bonus”, which may cause confusion for people trying to understand the code. For example, a 25% discount is equivalent to a 33% bonus. That is, “25% off” is the same as “33% more for free”.

The comment on [line 6 of Exponential.sol](#) says “fixed-decision” when it should say “fixed-precision”.

The comment on [line 83 of CToken.sol](#) describes borrowIndex as the accumulator of earned interest when it should be the accumulator of the earned interest rate.

Consider updating the comments appropriately.

Excessive Indirection

The [Exponential contract](#) represents a fixed-size decimal number with a uint that is scaled up so the smallest non-zero decimal value is internally represented by the number 1. However, it is also unnecessarily wrapped in a struct.

This has the advantage that the scaled values can have a unique Solidity type. On the other hand, this feature is used inconsistently throughout the code base and introduces a very large overhead. Many of the functions in the [Exponential](#) contract implement common mathematical operations on the Exp type when the equivalent functions already exist for the uint256 type. Most of the functions in the Exponential contract could be simplified or eliminated if the additional layer of indirection were removed. Additionally, many operations are complicated by mapping back and forth between the two representations.

Consider using the uint256 type to internally represent the fixed-size decimal numbers. Then, consider enforcing the existing Mantissa suffix convention to consistently indicate whether a given variable is scaled.

Low Severity

cToken Related Issues

cTokens might not be transferable

Currently cTokens can not be transferred if they are required to collateralize a loan, but there are no functions to check if users' cTokens are locked. This is not standard ERC20 behaviour and could be confusing to users.

Considering adding a related function to check if users' cTokens are transferable.

ERC20 double spend race condition

Due to CToken's inheritance of ERC20's approve function, it is vulnerable to the ERC20 approve and double spend front running attack. Briefly, an authorized spender could spend both allowances by front running an allowance-changing transaction. Consider implementing

OpenZeppelin's [decreaseAllowance](#) and [increaseAllowance](#) functions to help mitigate this.

ERC20 decimals size

The EIP20 specification [optionally defines a uint8 decimals field](#). However, the [corresponding field](#) in the CToken contract is a uint256. This is not compliant with the specification and may cause confusion when interacting with wallets and dApps. Consider setting the decimals type to uint8.



0 Address for Mints & Burns

Although not technically part of the EIP20 specification, it is common practice to use the zero address as the source for all Transfer events after minting, and as the destination for Transfers upon burning tokens. The Transfer events in functions [mintFresh](#) and [redeemFresh](#) use the address of the CToken contract instead. Consider using the zero address instead or informing users and developers of this feature.

Mixing concerns

The CToken contract performs three different categories of functions:

1. ERC20 operations including transferring tokens, checking balances and setting allowances
1. Administrative operations such as setting the comptroller, interest rate model and reserve rate
1. The borrowing, loaning, repaying and liquidating operations as well as their support functions

For simplicity and readability, consider splitting it into multiple contracts with distinct roles.

Admin Must Receive Reserves

The CToken administrator can call the [_reduceReserves function](#) to withdraw some of the reserves. However, the function requires that the administrator is the recipient address. This merges roles that should probably be distinct, particularly when the administrator is replaced with a decentralized governance process.

Consider having a separate recipient role, or allowing the administrator to choose the recipient in the function call.

Truncation-Related Issues

Throughout the compound contracts, truncation issues inherent to EVM operation result in some relatively unavoidable errors. These errors exist when minting cTokens, when redeeming cTokens for their underlying assets, and when liquidating another user's loan.

In the case of minting, the CToken.mintFresh function [calls divScalarByExpTruncate](#) to determine the number of cTokens the user will receive given their input of a certain number of underlying tokens. The result will be truncated if it is calculated to be some non-integer number of cToken units.

In the case of redeeming, the CToken.redeemFresh function also [calls divScalarByExpTruncate](#). The number of tokens to redeem will similarly be truncated to the next lowest integer value of underlying token units.

Finally, the Comptroller.liquidateCalculateSeizeTokens function [calls mulScalarTruncate](#). If the amount repaid is sufficiently small, the result will be rounded down to the next nearest integer value of indivisible token units.

In all cases, the user receives less than they theoretically should (either in terms of cTokens, or in terms of underlying tokens). However, the loss should never be more than 1 indivisible unit of whatever token the user is receiving. Even in the case of wBTC, where 1 indivisible unit is worth the

most out of any other token involved, the loss is only roughly 1% of a USD cent (at time of writing). This issue, unlike the issue of interest-free loans, does not hurt the protocol. Instead, the users take the brunt of the (very small) loss.

It should be pointed out that extremely small loans that are able to be liquidated may not actually be liquidated, since truncation could cause the liquidator to receive nothing, or far less than they'd theoretically receive. However, loans that are not liquidated because of this will eventually accrue enough interest that they can be profitably liquidated.

Partial Refactorization

The [getUtilizationAndAnnualBorrowRate function](#) of the WhitePaperInterestRateModel contract appears to be in a partially refactored state. The comments and choice of functions suggest that the [multiplier variable](#) is represented as a percentage (ie. the value 45 would imply a multiplier of 45%). However, instead of dividing by 100, [it is divided by 1e18](#). This is consistent with how the deployed contracts treat the multiplier variable. Nevertheless, it uses the functions intended for uint values to recreate the [mulExp](#) function without the rounding check. Consider using the [mulExp](#) function to scale the utilization rate instead of [mulScalar](#) and [divScalar](#) and update the comments to describe the correct code behaviour. In addition, consider stating the unit type in the [multiplier and baseRate comments](#).

Notes

Markets Can Become Insolvent

When the value of all collateral is worth less than the value of all borrowed assets, we say the market is *insolvent*. Compound does many things to reduce the risk of market insolvency, including: prudent selection of collateral-ratios, incentivizing third-party collateral liquidation, careful selection of which tokens are listed on the platform, etc. However, the risk of insolvency cannot be entirely eliminated, and there are numerous ways a market can become insolvent.

Here are five examples of things that could cause a market to become insolvent:

1. The price of the underlying (or borrowed) asset makes a big, quick move during a time of high network congestion — resulting in the market becoming insolvent before enough liquidation transactions can be mined.
2. The price oracle temporarily goes offline during a time of high market volatility. This could result in the oracle not updating the asset prices until after the market has become insolvent. In this case, there will never have been an opportunity for liquidation to occur.
3. The admin or oracle steals enough collateral that the market becomes insolvent.
4. Miners “steal” (by not paying interest) enough funds that the market eventually becomes insolvent. (See the “Interest-Free Loans” issue).

5. Administrators list an ERC20 token with a later-discovered bug that allows minting of arbitrarily many tokens. This bad token is used as collateral to borrow funds that it never intends to repay.

In any case, the effects of an insolvent market could be disastrous. It would mean that cToken contracts would effectively be running a fractional reserve. This could result in a “run on the bank”, with the last suppliers out losing their money.

This risk is not unique to Compound. *All* collateralized loans (even non-blockchain loans) have a risk of insolvency. However, it is important to know that this risk does exist, and that it can be difficult to recover from even a small dip into insolvency.

Reserves Increased Event

The [accrueInterest function of the CToken contract](#) accrues interest and increases the reserves. However, the `AccrueInterest` event that it emits does not include the amount by which the reserves have increased. Consider adding this value to the `AccrueInterest` event or creating a new `ReservesIncreased` event to match the existing `ReservesReduced` event.

Multiple Contracts Per File

The [ErrorReporter file](#) contains three independent contracts: `ComptrollerErrorReporter`, `TokenErrorReporter` and `OracleErrorReporter`. This does not follow the [Solidity style guide](#) and makes the code harder to read. Consider using a separate file for each contract.

Inconsistent NatSpec Usage

The docstrings of the contracts and functions are partially following the [Ethereum Natural Specification Format \(NatSpec\)](#). They use the tags sporadically. Consider adding the relevant missing tags to all contracts and functions.

Incorrect Code Comments

The comment on [line 906 of Comptroller.sol](#) uses the variable `newLiquidationDiscount` when it should use `newLiquidationIncentive`.

The comment on [line 298 of CToken.sol](#) states that the `transferVerify` function checks for under-collateralization. In fact, that check is performed at the start of the function with the `transferAllowed` hook.

The comment on [line 821 of CToken.sol](#) states that `redeemAmountIn` is the amount of cTokens to redeem but it is the amount of underlying.

The comments on [line 821-822 of CToken.sol](#) state that only one of `redeemTokensIn` or `redeemAmountIn` may be zero but in fact at least one must be zero and both may be zero.

The comment on [line 322 of Comptroller.sol](#) should state “Require tokens is non-zero or amount is also zero”

Consider updating the comments appropriately.

Require Statement Without Error Message

There is a require statement on [line 111 of CEther.sol](#) with no failure message. Consider adding a message to inform users in case of a revert.

Non-traditional Use of ReentrancyGuard

The [NatSpec documentation on ReentrancyGuard.sol](#) states:

If you mark a function nonReentrant, you should also mark it external.

However, the following functions in [CToken.sol](#) have the nonReentrant modifier and are NOT external:

exchangeRateCurrent (public)

mintInternal (internal)

redeemInternal (internal)

redeemUnderlyingInternal (internal)

borrowInternal (internal)

repayBorrowInternal (internal)

repayBorrowBehalfInternal (internal)

liquidateBorrowInternal (internal)

Typically, the nonReentrant modifier would be put on their external function counterparts in [CEther.sol](#) and [CErc20.sol](#). That said, the existing use of ReentrancyGuard *does* prevent reentry into the corresponding external functions, and so no changes are needed.

Unvetted Token Warning

It is important to note that if a malicious or poorly-designed token is added to Compound, it could allow someone to steal *all* funds entrusted to Compound. For example, if anyone can arbitrarily change the `totalSupply` or account balances of a listed ERC20 token faster than the price oracle can adjust the price, an attacker could use those newly minted tokens as collateral to borrow all Compound assets.

Consider making a formal list of properties that a “safe” token should and should not have, and be sure each new token is safe before listing it on Compound.

Transparent Proxy Pattern

The [Unitroller contract](#) uses a proxy pattern to redirect transactions to functions it does not recognize. However, it will not be able to proxy calls to functions with the same function signature as one of its own functions, including its inherited functions.

Consider using [the transparent proxy pattern](#) for increased generality.



Unused Return Value

The `getUtilizationAndAnnualBorrowRate` function of the `WhitePaperInterestRateModel` contract [returns the utilization rate](#) in addition to the borrow rate. However, the only function that calls it [discards the utilization rate](#). Consider removing the utilization rate from the return values.

collateralFactorMantissa Needs to be Set

In `Comptroller.sol`, `collateralFactorMantissa` [for each market is initialized implicitly to 0](#) until it is set within `_setCollateralFactor`. This means that whenever a new market is created, a second transaction must be submitted to set the collateral factor. If this doesn't happen, [account liquidity contributions for that market will be 0](#).

Consider Adding Syntax Highlighting to GitHub Repository

Consider [adding Solidity syntax highlighting](#) to the GitHub repository by putting the line `*.sol linguist-language=Solidity` in a `.gitattributes` file in the root of the repository.

This helps improve readability for users inspecting contract code on GitHub.

Accrue Zero Interest Event

In the [accrueInterest function of CToken](#), the `AccrueInterest` event is still emitted when there is no interest (because it was already accrued in this block). Consider checking the `accrualBlockNumber` before emitting the event (or indeed, executing the rest of the function).

Unexpected ERC20's

Within each of the `cToken` ERC20 contracts, there is no way to check for unexpected sends of ERC20 tokens to the contract. However, the ERC20 balance is used in the [exchangeRate calculation](#) via the variable `totalCash`, [which is equal to the ERC20 token balance](#).

This results in `cToken` values increasing with ERC20 sends, but importantly, it *does not* result in the value of borrows increasing, as `borrowIndex` is unaffected by total underlying balance.

Consider adding a state variable to track internal balances, which would help identify any unexpected ERC20 tokens.

Avoid Overloaded Functions

Within `Exponential.sol` there are two definitions for `mulExp()` ([here](#), and [here](#)). Consider changing the name of one of the functions for improved readability of code.

Use ABI Encoder

The [requireNoError function in the CEther contract](#) appends a string to another string one byte at a time. Consider commenting the operation for clarity and using `abi.encodePacked` for brevity.

Missing require

The [sanity check](#) in the `Comptroller _supportMarket` function confirms the passed `CToken` contract has an `isCToken` method, but it does not confirm that it returns `true`. Consider wrapping it in a `require` statement.

Default Visibility

The following state variables and constants are using the default visibility:

In `Comptroller`:

[closeFactorMinMantissa](#), [closeFactorMaxMantissa](#), [collateralFactorMaxMantissa](#), [liquidationIncentiveMinMantissa](#), [liquidationIncentiveMaxMantissa](#)

In `CToken`:

[borrowRateMaxMantissa](#), [reserveFactorMaxMantissa](#), [accountTokens](#),
[transferAllowances](#), [accountBorrows](#)

In `Exponential`:

[expScale](#), [halfExpScale](#), [mantissaOne](#)

For readability, consider explicitly declaring the visibility of all state variables.

Conclusion

No critical and two high severity issues were found. Some changes were proposed to follow best practices and reduce potential attack surface.