HashEye

// SMART CONTRACT SECURITY AUDIT

Compound

Security assessment by HashEye · prepared for Ethereum/EVM

HASHEYE AUDITED

PROJECT	Compound
PERFORMED BY	HashEye Security Review Team - Sergey, Brian
RESEARCH	Sergey, Brian
CLIENT	Ethereum/EVM
CATEGORY	Ethereum/EVM
PUBLISHED	February 1, 2020
REPORT ID	research-compound-2020-02-01-gotll8

This report was produced by HashEye's security review team under the HashEye review process, combining source review, deployment-state checks, and senior manual verification. Full report detail is available at hasheye.io/audits/research-compound-2020-02-01-gotll8.

Compound v2 Security Assessment April 8, 2019 Prepared For: Robert Leshner | Compound Labs robert@compound.finance Geoff Hayes | Compound Labs geoff@compound.finance Prepared By: Michael Colburn | HashEye michael.colburn@hasheye.io Evan Sultanik | HashEye evan.sultanik@hasheye.io Changelog: April 8, 2019: Public release March 22, 2019: Week 3 results delivered to Compound

Executive Summary Engagement Goals & Scope Coverage Project Dashboard Recommendations Summary Short Term Long Term Issues Addressed Since the First Assessment Findings Summary 14. Error propagation instead of reverting is inefficient and dangerous 15. Potential reentrancy from malicious tokens 16. Redundant data storage in the Comptroller can lead to desynchronization 17. Solidity compiler optimizations can be dangerous 18. Token migration results in orphaned balances 19. Race condition in the ERC20 approve function may lead to token theft 20. Unnecessary nonReentrant modifiers waste gas 21. Using CTokens as underlying assets may have unintended side effects 22. A malicious contract can reentrantly bypass administrative checks in the Comptroller 23. Variable shadowing between the Unitroller and Comptroller Appendix A. Vulnerability Classifications Appendix B: Code Quality Recommendations © 2019 HashEye

Executive Summary Between March 4th and March 22nd and again during the week of April 1st, 2019 HashEye assessed the smart contracts of the Compound v2 Ethereum codebase. Two engineers conducted this assessment over the course of eight person-weeks. HashEye had assessed the prior release of Compound in late July of 2018. The first week consisted of a high level architectural review. However, the extant code was consulted to fill in gaps and provide context for portions of the specification that were in flux. The second week initiated an in-depth assessment of release v2.1-Beta1 of the codebase. The third week concluding the general manual analysis of a slightly newer 1 release. The fourth week specifically focused on the contract upgrade mechanisms of yet 2 another slightly newer release. 3 Ten new findings were discovered, four of which are high severity. The majority are related to either untrusted calls to external contracts or gas-usage inefficiencies. Several bugs stem from logical errors and design decisions necessitated by Compound’s error reporting mechanism and avoidance of reverts. Additional code-quality recommendations that are not necessarily related to security have been added to Appendix B . 1 git commit 9e6e50320dacc93b9f9f834f5dffbaa8727550ad 2 git commit 4c93c9363d5f554717e5a880b74c052acaee7558 3 git commit b7475c1a7c04f8a558d21db672f82ba42cae531f © 2019 HashEye

Engagement Goals & Scope The goal of the engagement was to evaluate the security of the Compound v2 system and answer the following questions: ●Can attackers use leverage within the system to undermine the stability of the market? ●Can the system handle wildly fluctuating assets while maintaining a bounded increase/decrease percentage? ●Do the contracts calculate prices, swings, and basis points correctly? ●Does the new Comptroller contract pattern introduce any vulnerabilities in its interaction with the token markets? ●Is the Unitroller/Comptroller upgrade proxy pattern implemented correctly? Coverage We reviewed: ●Compound v2 whitepaper and protocol specification ●money-market v2.1-Beta1 ●money-market commit 4c93c9363d5f554717e5a880b74c052acaee7558 ●money-market commit b7475c1a7c04f8a558d21db672f82ba42cae531f This review included all Solidity smart contracts, JavaScript tests, as well as their requisite configuration files and environments. Contracts were reviewed for common Solidity flaws, such as integer overflows, re-entrancy vulnerabilities, and unprotected functions. Furthermore, contracts were reviewed with special consideration for high-level logical flaws and unhandled edge cases in the interaction between the new Comptroller and CToken contracts. The contract upgradability mechanism implemented in commit b7475c1a7c04f8a558d21db672f82ba42cae531f was specifically analyzed during the final week of the assessment. Special care was placed in checking for variable shadowing, function shadowing, initialization, and slot ordering errors. © 2019 HashEye

Project Dashboard Application Summary Name Compound v2 Type Money Market, ERC20 Token, and Protocol Platform Solidity Engagement Summary Dates March 4 th through March 22 nd , 2019 April 1 st through April 5 th , 2019 Method Whitebox Consultants Engaged 2 Level of Effort 8 person-weeks Vulnerability Summary Total High Severity Issues 4 ▯▯▯▯ Total Medium Severity Issues 0 Total Low Severity Issues 0 Total Informational Severity Issues 4 ▯▯▯▯ Total Issues of Undetermined Severity 2 ▯▯ Total 10 Category Breakdown Access Controls 2 ▯▯ Data Validation 3 ▯▯▯ Denial of Service 1 ▯ Error Reporting 1 ▯ Timing 1 ▯ Undefined Behavior 2 ▯▯ Total 10 © 2019 HashEye

Recommendations Summary Short Term ☐ Carefully employ error handling. The pattern used for error propagation and handling is inefficient and dangerous. Be diligent about usage of functions that return an error state. ☐ Fix the potential for reentrancy. Either adhere to the checks-effects-interactions pattern or use reentrancy guards. ☐ Document redundant data storage in the Comptroller. Improve source code comments and sanity checks to reinforce the importance of keeping the data structures synchronized. ☐ Evaluate the costs and benefits of Solidity optimizations. Measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug. ☐ Add a mitigation for orphaned balances. Consider adding an admin-only function to the CToken contract to withdraw unexpected or unsupported ERC20 tokens. ☐ Implement a mitigation for the ERC20 race condition. For example, implement the increaseApproval and decreaseApproval functions of OpenZeppelin's ERC20 implementation. ☐ Remove unnecessary nonReentrant modifiers. Weigh the value of avoiding view functions against the cost of higher gas overhead. Consider refactoring the checks-effects-interactions pattern wherever possible. ☐ Do not allow CTokens as an underlying asset of another CToken. Consider the implications of this edge case. ☐ Fix the admin or initializing check. Either convert the adminOrInitializing function to a modifier, or pass in the sender address to verify as a function argument. ☐ Document storage variable shadowing between the Unitroller and Comptroller. All future implementations of the Comptroller must extend from UnitrollerAdminStorage (via ComptrollerV1Storage) in order to preserve the storage layout. New Comptroller instances must always be constructed from the same account that is listed as admin in the Unitroller. © 2019 HashEye

Long Term ☐ Consider abandoning the error-handling pattern. This will serve to make the code more maintainable, succinct, and less expensive. ☐ Consider refactoring redundant storage in the Comptroller. Carefully measure the gas benefit of reduced iterations versus the cost of additional storage. Consider removing one of the data structures so that there is no chance of desynchronization. ☐ Monitor the development and adoption of Solidity compiler optimizations. Continually assess the maturity of Solidity optimizations. ☐ Document procedures for changes to external contracts. What happens when the token backing an asset performs a migration? ☐ Monitor the progress of alternative reentrancy protections. For example, EIP 1153 introduces a much less expensive ephemeral storage opcode that could be used for reentrancy protection. ☐ Consider whether CTokens should ever be listed. If so, investigate potential side effects and document procedures for updating their parameters. Also, clearly document the necessary criteria for a token to be listed on Compound. This will be especially important if the protocol transitions to a decentralized governance model in the future. ☐ Avoid variable shadowing between proxies and implementations. For example, the ZeppelinOS implementation of the delegatecall proxy pattern uses a custom storage slot to store the admin address in order to avoid such collisions; the implementation can safely use storage starting at slot zero. Alternatively, consider the contract migration upgrade pattern . © 2019 HashEye

Issues Addressed Since the First Assessment This is a summary of issues uncovered during the first assessment by HashEye of Compound Release Candidate 1.1 in August of 2018. The table lists whether each finding has been addressed in version 2.1 of the protocol. Only findings related to the protocol and smart contracts are included. # Title Severity Addressed? 1 CarefulMath error handling is inefficient and dangerous Informational No 2 Authentication check replication Informational No 3 ERC20 interface undefined behavior High Yes 4 Frontrunning can be used to weaken collateral requirements Medium In theory, yes, but it is unclear if the new model can react fast enough 8 MoneyMarket should be pausable Informational Yes, in development 12 Missing error-handling defaults Informational No © 2019 HashEye

Findings Summary # Title Type Severity 14 Error propagation instead of reverting is inefficient and dangerous Error Reporting Informational 15 Potential reentrancy from malicious tokens Data Validation High 16 Redundant data storage in the Comptroller can lead to desynchronization Data Validation Informational 17 Solidity compiler optimizations can be dangerous Undefined Behavior Undetermined 18 Token migration results in orphaned balances Access Controls High 19 Race condition in the ERC20 approve function may lead to token theft Timing High 20 Unnecessary nonReentrant modifiers waste gas Denial of Service Informational 21 Using CTokens as underlying assets may have unintended side effects Data Validation Undetermined 22 A malicious contract can reentrantly bypass administrative checks in the Comptroller Access Controls High 23 Variable shadowing between the Unitroller and Comptroller Undefined Behavior Informational © 2019 HashEye

14. Error propagation instead of reverting is ineficient and dangerous Severity: InformationalDifficulty: Low Type: Error ReportingFinding ID: Compound-TOB-014 Target: All smart

contracts Description The contracts in Compound version 2.1 retain the unusual error handling pattern from the prior version in which errors are reported as a second return argument rather than immediately throwing an assertion.

function sub(uint a, uint b) internal pure returns (Error, uint) { if (b ⩽ a) { return (Error.NO_ERROR, a - b); } else { return (Error.INTEGER_UNDERFLOW, 0); } } Figure 14.1: Error handling in the CarefulMath subtraction function. This can be very useful and informative for debugging purposes, and allows callers the opportunity to gracefully handle certain errors. However, it suffers from three major shortcomings: 1.This implementation requires the caller to check for the error (q.v. Figure 14.2). This not only bloats the calling usages, but allows for a careless caller to mishandle the error at the callsite.

(Error err0, uint blockDelta) = sub(blockEnd, blockStart); if (err0 ≠ Error.NO_ERROR) { return (err0, 0); } Figure 14.2: Example usage of the CarefulMath subtraction function. 2.Additional control flow logic and branching is required at each callsite, drastically increasing the complexity of the code, and thereby the number of resulting EVM opcodes, and thereby the gas cost of executing the code. 3.Using the Checks-Effects-Interactions pattern to avoid reentrancy attacks becomes difficult if the interactions return an error instead of reverting, since all of the effects would have to be unwound manually. © 2019 HashEye Compound v2 Security Assessment | 9

Our calculations suggest that each call to a function using this pattern incurs up to three times the gas of an alternative that uses a standard require statement. This can constitute a significant gas expenditure. Exploit Scenario A future refactor causes a function to be called without checking for its resulting error condition. Recommendation In the short term, be extremely careful and diligent about usage of functions that return an error state. In the long term, consider abandoning this error-handling pattern. Not only will this serve to make the code more maintainable, succinct, and less expensive, but it will enable you to use the canonical “ using SafeMath for uint256; ” idiom that will ensure that all arithmetic will be handled using SafeMath. This should be relatively straightforward since it appears that all usages of CarefulMath functions either blindly pass on errors to their caller or ignore the specific error. © 2019 HashEye Compound v2 Security Assessment | 10

15. Potential reentrancy from malicious tokens Severity: HighDifficulty: High Type: Data ValidationFinding ID: Compound-TOB-015 Target: CToken.sol and MoneyMarket.sol Description Functions that make external calls to underlying token assets fail to use the “Checks-Effects-Interactions” pattern . This is presumably related to Compound-TOB-014 , since properly adhering to the pattern without using a revert would make the code more complex and significantly increase the gas cost. For example, Figure 15.1 is an excerpt from the withdraw function in MoneyMarket.sol . ////////////////////////// // EFFECTS & INTERACTIONS // (No safe failures beyond this point)

// We ERC-20 transfer the asset into the protocol (note: pre-conditions already checked above) err = doTransferOut(asset, msg.sender, localResults.withdrawAmount); if (err ≠ Error.NO_ERROR) { // This is safe since it's our first interaction and it didn't do anything if it failed return fail(err, FailureInfo.WITHDRAW_TRANSFER_OUT_FAILED); }

// Save market updates market.blockNumber = getBlockNumber(); market.totalSupply = localResults.newTotalSupply; market.supplyRateMantissa = localResults.newSupplyRateMantissa; market.supplyIndex = localResults.newSupplyIndex; market.borrowRateMantissa = localResults.newBorrowRateMantissa; market.borrowIndex = localResults.newBorrowIndex;

// Save user updates localResults.startingBalance = supplyBalance.principal; // save for use in `SupplyWithdrawn` event supplyBalance.principal = localResults.userSupplyUpdated; supplyBalance.interestIndex = localResults.newSupplyIndex; Figure 15.1: Code using a pattern that allows reentrancy. The call to doTransferOut will interact with the underlying asset contract. That contract can do whatever it wants, including calling back into the Compound contracts with a reentrancy attack. This all happens before the effects of the withdrawal, like the supply changes, are recorded. Unfortunately, it is non-trivial to reorder this implementation to conform to the standard checks-effects-interactions pattern, since an error in doTransferOut would require the nine changes to market , localResults , and supplyBalance to be unwound to the previous © 2019 HashEye Compound v2 Security Assessment | 11

values, which would also have to be cached. This would result in a significant gas expenditure. Exploit Scenario A malicious token is approved as an asset, either surreptitiously or through a future distributed governance mechanism. That token can reentrantly call back into the Compound contracts before the effects of the previous action have been recorded, causing the Compound contracts to be in an invalid state. Recommendation In the short term, fix the potential of reentrancy by adhering to the checks-effects-interactions pattern. In the long term, consider

abandoning the complex error reporting pattern in favor of standard require statements that are guaranteed to revert.

16. Redundant data storage in the Comptroller can lead to desynchronization Severity: InformationalDifficulty: High Type: Data ValidationFinding ID: Compound-TOB-016 Target: Comptroller.sol Description The accountMembership and accountAssets members of the Comptroller contract store redundant information. The correctness of portions of the code is predicated on those members remaining in sync, since sometimes data is validated against one and not the other. This is a known issue, since a code comment suggests that it is an optimization for alleviating the need to iterate over members. Exploit Scenario In a future refactor, checks like this requirement in the Comptroller are removed since it is assumed that accountMembership and accountAssets will always be synchronized: // paranoid check that we actually found something in the list // this will always be true and maybe we can delete the if check // if confident in formal verification of the assets you are in system require(assetIndex < len); An attacker causes this assumption to be invalidated, e.g. , through a reentrancy attack like that of Compound-TOB-015 . This will allow the attacker to underflow accountAssets[msg.sender].length , gaining the ability to overwrite all of storage. Recommendation In the short term, improve source code comments to reinforce the importance of keeping the data structures synchronized. Consider adding more sanity checks like the above require example. In the long term, carefully measure the gas benefit of reduced iterations versus the cost of additional storage. Consider removing one of the data structures so that there is no chance of desynchronization.

17. Solidity compiler optimizations can be dangerous Severity: UndeterminedDifficulty: Low Type: Undefined BehaviorFinding ID: Compound-TOB-017 Target: truffle.js

Description Compound has enabled optional compiler optimizations in Solidity. There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed . Solidity compiler optimizations are disabled by default. It is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised. High-severity security issues due to optimization bugs have occurred in the past . A high-severity bug in the emscripten-generated solc-js compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6 . A compiler audit of Solidity from November, 2018 concluded that the optional optimizations may not be safe . Moreover, the Common Subexpression Elimination (CSE) optimization procedure is “implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function.” Similar code in other large projects have resulted in bugs. There are likely latent bugs related to optimization, and/or new bugs that will be introduced due to future optimizations. Exploit Scenario A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to solc-js —causes a security vulnerability in the Compound contracts. Recommendation In the short term, measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug. In the long term, monitor the development and adoption of Solidity compiler optimizations to assess its maturity.

18. Token migration results in orphaned balances Severity: HighDifficulty: High Type: Access ControlsFinding ID: Compound-TOB-18 Target: CToken.sol Description The Compound protocol establishes money markets for ERC20 tokens. For a variety of reasons, including security issues or to support new features, these underlying tokens may perform a migration to a new contract in the future. The token balances from the old contract will be transferred as-is to the new contract. All funds deposited on Compound will remain credited to the corresponding CToken . However, the functionality of the CToken will still refer to the old token address; borrows and redemptions will be impossible to fulfill. Exploit Scenario Compound has an established market for ZRX . Alice deposits 1,000 ZRX into the Compound ZRX market. The 0x team decides to migrate to a new version of the token contract to add support for a new feature. Alice wishes to withdraw some of her ZRX held in Compound but is unable to as the CToken contract for the market has no functionality to interact with the new ZRX contract. Recommendation In the short term, consider adding an admin-only function to the CToken contract to withdraw unexpected or unsupported ERC20 tokens. In the event of a migration, the admin will then be able to manually redistribute tokens to Compound users. In the long term, document procedures for handling changes in the entry point of any external contracts. References How contract migration works

19. Race condition in the ERC20 approve function may lead to token thet Severity: HighDifficulty: High Type: Timing Finding ID: Compound-TOB-019 Target: CToken.sol Description Compound

markets, implemented as CToken contracts, are themselves ERC20 compliant. A known race condition in the ERC20 standard, on the approve function, could lead to the theft of tokens. The ERC20 standard describes how to create generic token contracts. Among others, an ERC20 contract defines these two functions: •transferFrom(from, to, value) •approve(spender, value)

These functions give permission to a third party to spend tokens. Once the function approve(spender, value) has been called by a user, spender can spend up to value of the user's tokens by calling transferFrom(user, to, value). This schema is vulnerable to a race condition when the user calls approve a second time on a spender that has already been allowed. If the spender sees the transaction containing the call before it has been mined, then the spender can call transferFrom to transfer the previous value and still receive the authorization to transfer the new value. Exploit Scenario 1.Alice calls approve(Bob, 1000) . This allows Bob to spend 1,000 tokens. 2.Alice changes her mind and calls approve(Bob, 500). Once mined, this will decrease to 500 the number of tokens that Bob can spend. 3.Bob sees Alice's second transaction and calls transferFrom(Alice, X, 1000) before approve(Bob, 500) has been mined. 4.If Bob's transaction is mined before Alice's, 1,000 tokens will be transferred by Bob. Once Alice's transaction is mined, Bob can call transferFrom(Alice, X, 500) . Bob has transferred 1,500 tokens, contrary to Alice's intention. Recommendations While this issue is known and can have a severe impact, there is no straightforward solution.

One workaround is to use two non-ERC20 functions allowing a user to increase and decrease the approve (see increaseApproval and decreaseApproval of StandardToken.sol#L63-L98). Another workaround is to forbid a call to approve if all the previous tokens are not spent by adding a require to approve. This prevents the race condition but it may result in unexpected behavior for a third party. require(_approvals[msg.sender][guy] == 0) This issue is a flaw in the ERC20 design. It cannot be fixed without modifications to the standard. It must be considered by developers while writing code.

20. Unnecessary nonReentrant modifiers waste gas Severity: InformationalDifficulty: Low Type: Denial of Service Finding ID: Compound-TOB-020 Target: CToken.sol Description Several functions within the CToken contract have been protected with OpenZeppelin's nonReentrant modifier from ReentrancyGuard . This alone is fine, since it is currently the second-best mitigation against reentrancy, after the checks-effects-interactions pattern. However, some of the functions (e.g. , borrowBalanceCurrent) could be declared as a view (see Appendix B for a comprehensive list). With Solidity 0.5.0 and newer, all external calls 4 from a view are compiled with the STATICCALL opcode introduced in Byzantium, which prevents reentrant storage modifications. Exploit Scenario A user unnecessarily spends extra gas or exhausts the block gas limit by interacting with the Compound v2 contracts. Recommendations In the short term, weigh the value of avoiding view functions (presumably, to enable unit test harnesses) against the cost of higher gas overhead. In the long term, monitor the progress of EIP 1153 , which would introduce a new, much less expensive ephemeral storage opcode that could be used for reentrancy protection. Also consider refactoring to the checks-effects-interactions pattern wherever possible. 4 although the feature appears to have been silently included as early as Solidity 0.4.24

21. Using CTokens as underlying assets may have unintended side efects Severity: UndeterminedDifficulty: Undetermined Type: Data Validation Finding ID: Compound-TOB-021 Target: CToken.sol Description Compound can support a market for any ERC20 compatible token. These markets are implemented as CToken contracts, which also implement the ERC20 interface. However, using a CToken as the underlying asset of a Compound market may lead to unintended side effects. Exploit Scenario A market exists for the ABC token, with associated CToken cABC . Likewise, a secondary market is created backed by cABC , with associated CToken ccABC . These markets both track approximately the same asset. However, their collateral factors (via the Comptroller), interest rate model and other variables are managed independently. They may deviate unintentionally over time. Recommendations In the short term, do not enable markets which use a CToken as the underlying asset. In the long term, consider whether CTokens should ever be listed and if so, investigate potential side effects and document procedures for updating their parameters. Also, clearly document the necessary criteria for a token to be listed on Compound. This will be especially important if the protocol transitions to a decentralized governance model in the future.

22. A malicious contract can reentrantly bypass administrative checks in the Comptroller Severity: HighDifficulty: High Type: Access Controls Finding ID: Compound-TOB-022 Target: Comptroller.sol Description The adminOrInitializing() function checks if msg.sender is the comptroller implementation and if the transaction originated from the admin account. /** * @dev Check that caller is admin or this contract is initializing itself as * the new implementation. *

There should be no way to satisfy msg.sender == comptrollerImplementaiton * without tx.origin also being admin, but both are included for extra safety */ function adminOrInitializing () internal view returns (bool) { bool initializing = (msg .sender == comptrollerImplementation && //solium-disable-next-line security/no-tx-origin tx .origin == admin); bool isAdmin = msg .sender == admin; return isAdmin || initializing; } However, msg.sender will always be equal to comptrollerImplementation , because adminOrInitializing is declared internal (and can thereby only ever be called from other functions in the Comptroller) and it is a function, not a modifier. Therefore, initializing will be true as long as the transaction originated from the admin account. Exploit Scenario The admin account executes a transaction that eventually calls a malicious external contract (e.g. , a malicious price oracle or underlying asset token). The malicious contract reentrantly calls a privileged function within the Comptroller (e.g. , to set the close factor or

change the price oracle). The call to adminOrInitializing() will return true , allowing the transaction to succeed. Recommendations In the short term, there are two potential mitigations: 1.Convert adminOrInitializing from a function to a modifier so that msg.sender is preserved; or 2.Pass msg.sender as a “ sender ” argument into adminOrInitializing and perform the validation checks against sender instead of msg.sender . Note that the first option will require reverting on failure, deviating from the current error reporting scheme. In the long term, reevaluate the need for the current error reporting scheme in deference to these sorts of subtle bugs and the relative safety of patterns like modifiers that revert on error.

23. Variable shadowing between the Unitroller and Comptroller Severity: InformationalDifficulty: Informational Type: Undefined Behavior Finding ID: Compound-TOB-023 Target: Comptroller.sol , ComptrollerStorage.sol , and Unitroller.sol Description Since the Comptroller extends from ComptrollerV1Storage which extends from UnitrollerAdminStorage , all of the storage variables and functions defined in UnitrollerAdminStorage will be shadowed in the Unitroller proxy contract by the same variables and functions in the Comptroller. If a user calls one such function (i.e. , comptrollerImplementation() , admin() , pendingAdmin() , and pendingComptrollerImplementation()) on the Unitroller contract, the Unitroller’s implementation will be executed. However, if a user calls one such function directly on the Comptroller, the Comptroller’s implementation will be executed. This finding has Informational severity because the shadowed functions do not change contract state. Exploit Scenario A future refactor or upgrade causes the Comptroller to have a different admin account address than the Unitroller. Recommendations In general, try to avoid state variables in the proxy, particularly public ones. In the short term, document this behavior. All future implementations of the Comptroller must extend from UnitrollerAdminStorage (via ComptrollerV1Storage) in order to preserve the storage layout. New Comptroller instances must always be constructed from the same account that is listed as admin in the Unitroller. In the long term, consider switching to a different pattern that does not require the Comptroller to shadow the proxy’s storage variables. For example, the ZeppelinOS implementation of the delegatecall proxy pattern uses a custom storage slot to store the admin address in order to avoid such collisions; the implementation can safely use storage starting at slot zero. Alternatively, consider the contract migration upgrade pattern . The inheritance hierarchy of the contracts can be analyzed by running slither --print inheritance-graph .

This produces a Graphviz file called contracts.dot which can be converted to a PDF via dot -Tpdf contracts.dot -o contracts.pdf

Appendix A. Vulnerability Classifications Vulnerability Classes Class Description Access Controls Related to authorization of users and assessment of rights Auditing and Logging Related to auditing of actions or logging of problems Authentication Related to the identification of users Configuration Related to security configurations of servers, devices or software Cryptography Related to protecting the privacy or integrity of data Data Exposure Related to unintended exposure of sensitive information Data Validation Related to improper reliance on the structure or values of data Denial of Service Related to causing system failure Documentation Related to documentation accuracy Error Reporting Related to the reporting of error conditions in a secure fashion Patching Related to keeping software up to date Session Management Related to the identification of authenticated users Timing Related to race conditions, locking or order of operations Undefined Behavior Related to undefined behavior triggered by the program Severity Categories Severity Description Informational The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth Undetermined The extent of the risk was not determined during this engagement Low The risk is relatively small or

is not a risk the customer has indicated is important © 2019 HashEye Compound v2 Security Assessment | 24

Medium Individual user’s information is at risk, exploitation would be bad for client’s reputation, moderate financial impact, possible legal implications for client High Large numbers of users, very bad for client’s reputation, or serious legal or financial implications Difficulty Levels Difficulty Description Undetermined The difficulty of exploit was not determined during this engagement Low Commonly exploited, public tools exist or can be scripted that exploit this flaw Medium Attackers must write an exploit, or need an in-depth knowledge of a complex system High The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue © 2019 HashEye Compound v2 Security Assessment | 25

Appendix B: Code Quality Recommendations The below issues do not pose a direct security risk to Compound, but can still be security-relevant and should be addressed. ●The allowance local variable in the transferTokens function of CToken shadows a function of the same name. ●The mantissaOne and mantissaOneTenth constants in Exponential.sol are not currently used. ●Use error constants instead of constant literals wherever possible. For example, at the beginning of the mintFresh function in CToken.sol , the allowed result from comptroller.mintAllowed should be compared to Error.NO_ERROR instead of 0 . ●Many functions do not modify contract state and can therefore be declared as view or pure . For example, this appears to be the case for all of the *Verify functions in Comptroller.sol . ●A significant number of functions can be declared external , which can greatly reduce gas costs for the caller. This is because external functions can read arguments directly from the calldata, while public functions must first copy the arguments to memory . Potential candidates include: ○In CToken.sol ■exchangeRateStored ■mint ■redeem ■borrowAsset ■repayBorrow ■repayBorrowBehalf ■borrowBalanceCurrent ■liquidateBorrow ■As well as several of the administrative _* functions ○In Comptroller.sol ■mintAllowed ■borrowAllowed ■repayBorrowAllowed ■liquidateBorrowAllowed ■seizeAllowed ■getAccountLiquidity ■liquidateCalculateSeizeTokens ■As well as several of the administrative _* functions ●Commit 454aaca , which added reentrancy protections, included this note in its commit message: EIP 1283 defines a new refund for set and set-back storage changes. We should probably upgrade our re-entry guard to use a process that mimics this as it's now very cheap. © 2019 HashEye Compound v2 Security Assessment | 26

Unintended consequences of EIP 1283 were the cause of the recent Constantinople hard-fork postponement . It is unlikely that EIP 1283—or any other EIP that permits gas refunds, for that matter—will ever be approved. Instead, we recommend tracking the progress of EIP 1153 , which proposes an alternative reentrancy protection mechanism. See the recommendations in Compound-TOB-020 for more information. © 2019 HashEye Compound v2 Security Assessment | 27