Compound Protocol

Security Assessment

August 16, 2019

Prepared For:
Geoff Hayes | *Compound Labs*
geoff@compound.finance

Robert Leshner | *Compound Labs*
robert@compound.finance

Prepared By:
Evan Sultanik | *Trail of Bits*
evan.sultanik@trailofbits.com

Robert Tonic | *Trail of Bits*
robert.tonic@trailofbits.com

Executive Summary

Between August 12th and 16th, 2019, Trail of Bits assessed the smart contracts of the Compound Protocol Ethereum codebase that is poised to be open-sourced. Two engineers conducted this assessment over the course of two person-weeks. Trail of Bits had assessed previous releases of Compound in March 2019 and July of 2018. This included an in-depth assessment of the codebase, as of git commit `f385d71`. The assessment emphasized errors introduced since the last assessment, vulnerabilities related to minting and transfer of assets, edge cases related to reentrancy, problems with error handling, and operational security issues.

Trail of Bit discovered five new findings, one of which is High severity. The majority are related to error reporting and data validation, particularly when interacting with external contracts. The majority of security findings from the previous security assessment of the Compound Protocol version 2.1 release still persist. These are summarized below in the Issues Addressed Since Previous Assessments section. Trail of Bits has included additional code-quality recommendations that are not necessarily related to security to Appendix B.

Engagement Goals & Scope

The goal of the engagement was to evaluate the security of the Compound Protocol system and answer the following questions:

- Have the security findings from previous assessments been addressed?
- 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 all of the smart contracts, tests, and off-chain code included in the `compound-protocol` git repository as of commit `f385d71983ae5c5799faae9b2dfea43e5cf75262.`

This review included all Solidity smart contracts, JavaScript tests, and their requisite configuration files and environments.

Trail of Bit reviewed the contracts 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 `Comptroller` and `CToken` contracts. Special care was taken in checking for variable shadowing, function shadowing, initialization, and slot ordering errors in the contract upgradability mechanism.

Project Dashboard

Application Summary

Name	Compound Protocol Git Commit `f385d71`
Type	Money Market, ERC20 Token, and Protocol
Platform	Solidity

Engagement Summary

Dates	August 12th through August 16nd, 2019
Method	Whitebox
Consultants Engaged	2
Level of Effort	2 person-weeks

Vulnerability Summary

Total High Severity Issues	1	■
Total Medium Severity Issues	1	■
Total Low Severity Issues	2	■■
Total Informational Severity Issues	0	
Total Issues of Undetermined Severity	1	■
Total	5	

Category Breakdown

Data Validation	2	■■
Documentation	1	■
Error Reporting	2	■■
Total	5	

Recommendations Summary

The following is a summary of the recommendations resulting from the security findings discovered during this assessment. It is followed by a summary of the latent recommendations from the previous assessment during the Spring of 2019. The status of the previous findings are discussed in further detail in the following section.

Short Term

❑ **Document the maximum error code size.** Error codes greater than or equal to 2080 will corrupt error messages.

❑ **Reevaluate the need for alphabetizing FailureInfo codes.** Only append to these enumerations to ensure the `Error` enumeration order remains unchanged.

❑ **Fix oracle validation.** Enforce the `isPriceOracle` requirement when setting the price oracle.

❑ **Fix the documentation on upper-bounds error checks.** Disambiguate the usage of `-1` versus `uint(-1)`. Ensure documentation is accurate throughout the codebase.

❑ **Ensure access controls are properly implemented.** The `setSeizeAllowed` function in the `BoolComptroller` is currently public and unprotected, allowing an attacker to break the `seize` functionality.

Long Term

❑ **Formally verify error code bounds.** Use a verifier like Manticore or Certora to ensure that `requireNoError` cannot be called with a value greater than or equal to 2080.

❑ **Thoroughly document the procedure for adding new error codes.** Instruct developers to only appending new `FailureInfo` values. Ensure errors are compared against the enumerations in `ErrorReporter`, and not to literals.

❑ **Add tests for oracle validation.** Ensure test coverage includes both positive and negative tests of success.

❑ **Ensure testing reflects the documentation.** The documentation and implementation should be both correct and consistent.

❑ **Improve test coverage for the `BoolComptroller`.** Check that unprivileged users cannot call functions like `setSeizeAllowed`.

Short Term Recommendations from Previous Assessments

❑ **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.

Long Term Recommendations from Previous Assessments

❑ **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.

Issues Addressed Since Previous Assessments

Tables in this section list whether each finding has been addressed in the latest version of the protocol assessed during this engagement. Only findings related to the protocol and smart contracts are included.

August 2018 review of Compound RC 1.1

This is a summary of issues uncovered during the first assessment by Trail of Bits in August of 2018 of Compound Release Candidate 1.1.

#	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, implemented in Compound V2's Price Oracle
12	Missing error-handling defaults	Informational	No

April 2019 review of Compound 2.0

This is a summary of issues uncovered during the assessment by Trail of Bits of Compound V2 in April 2019.

#	Title	Severity	Addressed?
14	Error propagation instead of reverting is inefficient and dangerous	Informational	No, the error message propagation pattern is still present, hindering use of the checks-effects-interactions pattern

15	Potential reentrancy from malicious tokens	High	Yes, external calls now revert on failure
16	Redundant data storage in the Comptroller can lead to desynchronization	Informational	No, the `accountMembership` and `accountAssets` members of the `Comptroller` contract still store redundant information
17	Solidity compiler optimizations can be dangerous	Undetermined	No, compiler optimizations are still enabled in the Truffle build configuration
18	Token migration results in orphaned balances	High	No, there does not appear to be any remedy if the ERC20 token contract of an underlying asset is migrated to a new address
19	Race condition in the ERC20 approve function may lead to token theft	High	This is a problem inherent in the ERC20 standard itself. It has been mitigated by Compound to the extent possible
20	Unnecessary non-Reentrant modifiers waste gas	Informational	Monitoring for better alternatives (see further recommendations related to EIP1153 in the Code Quality Appendix)
21	Using CTokens as underlying assets may have unintended side effects	Undetermined	No, there is no check in `CErc20` to determine whether the underlying asset is itself a `CToken`
22	A malicious contract can re-entrantly bypass administrative checks in the Comptroller	High	No, the `adminOrInitializing()` function is still vulnerable to malicious external contracts
23	Variable shadowing between the Unitroller and Comptroller	Informational	No, the same proxy pattern exists, and the behavior does not appear to have been documented

Findings Summary

#	Title	Type	Severity
24	Large error codes will invalidate error messages	Error Reporting	Low
25	Failure information changes could cause on- and off-chain repercussions	Error Reporting	Medium
26	Lack of newOracle validation when setting a price oracle	Data Validation	High
27	Invalid documentation regarding upper-bounds error checks	Documentation	Low
28	Missing tests and functionality for checking senders when setting allowSeize	Data Validation	Undetermined

24. Large error codes will invalidate error messages

Severity: Low	Difficulty: Low
Type: Error Handling	Finding ID: Compound-TOB-024
Target: `contracts/CEther.sol`	

Description

The internal function `requireNoError` is used throughout the `CEther` token as a utility to generate human-readable log messages, containing both the error message and the error number. The error number is calculated as follows:

```
        fullMessage[i+2] = byte(uint8(48 + ( errCode / 10 )));
```

Figure 24.1: Line in which the first byte of the error code is appended to the error message. Bits will be truncated from the cast if `errCode` is too large.

The cast to `uint8` will truncate all but the eight least significant bits of the argument. Therefore, any `errCode` greater than or equal to (256 - 48) × 10 = 2080 will cause an overflow, and bits will be lost. Moreover, 2080 and any higher `errCode` that is a multiple of 2560 from 2080 will cause `fullMessage[i+2]` to be zero.

Exploit Scenario

A future refactor of the code causes an error code to be set to 2080. When the error occurs, the message generated by `requireNoError` will contain a zero byte at the start of the error code. Any code that treats the error message as a null-terminated string will interpret an incorrect code.

Recommendation

In the short term, document the requirement that all error codes be strictly less than 2080.

In the long term, add formal properties (*e.g.*, through Manticore or Certora) to verify that `requireNoError` cannot be called with a value greater than or equal to 2080.

25. Failure information changes could cause on- and off-chain repercussions

Severity: Medium　　Difficulty: Low
Type: Error Reporting　　Finding ID: Compound-TOB-025
Target: `ErrorReporter.sol, various contracts`

Description
Compound uses the `ErrorReporter` contract to generate errors throughout the system. However, documentation within the `ErrorReporter` indicates that `FailureInfo` is kept in alphabetical order. If new information codes are added, they will be inserted arbitrarily into the enumeration, based on alphabetical order.

Keeping the alphabetical order could lead to misinterpretation of failure codes by third-party systems interacting with Compound.

```
   /*
     * Note: FailureInfo (but not Error) is kept in alphabetical order
     *       This is because FailureInfo grows significantly faster, and
     *       the order of Error has some meaning, while the order of FailureInfo
     *       is entirely arbitrary.
     */
    enum FailureInfo {
        ACCEPT_ADMIN_PENDING_ADMIN_CHECK,
        ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED,
        ACCRUE_INTEREST_BORROW_RATE_CALCULATION_FAILED,
        ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULATION_FAILED,
        ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED,
        ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED,
        ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CALCULATION_FAILED,
        BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
        BORROW_ACCRUE_INTEREST_FAILED,
        BORROW_CASH_NOT_AVAILABLE,
        BORROW_FRESHNESS_CHECK,
        BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED,
        BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED,
        BORROW_MARKET_NOT_LISTED,
...
```

Figure 25.1: A snippet of the `ErrorReporter.sol` contract, displaying the documentation and enumeration of failure information codes.

On top of potentially misleading third-party systems that rely on the `ErrorReporter`, areas of the Compound system compare returned errors to literals, instead of the `ErrorReporter` enumeration constant. If the order of the enumerations ischanged, this could affect the validity of the comparisons, breaking disparate portions of the system.

```
       /* We calculate the number of collateral tokens that will be seized */
        (uint amountSeizeError, uint seizeTokens) =
comptroller.liquidateCalculateSeizeTokens(address(this), address(cTokenCollateral),
repayAmount);
        if (amountSeizeError != 0) {
            return failOpaque(Error.COMPTROLLER_CALCULATION_ERROR,
FailureInfo.LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED, amountSeizeError);
        }
```

Figure 25.2: A snippet of the `CToken.sol` *contract not referring to the* `ErrorReporter` *enumerations, but to literals instead.*

Exploit Scenario

A developer adds a value to `FailureInfo`, following alphabetical order formatting, as instructed. This results in the `FailureInfo` codes that third parties depend on being incorrect when the developer's changes are deployed.

A developer prepends a value to the `Error` enumeration. This results in every comparison relying on the literal `0` representing `NO_ERROR` being incorrect.

Recommendation

In the short term, avoid keeping alphabetical order for `FailureInfo` codes, and only append to these enumerations.

In the long term, change the documentation to instruct appending new `FailureInfo` values, instead of inserting them alphabetically. Ensure errors are compared against the enumerations in `ErrorReporter`, and not to literals.

26. Lack of newOracle validation when setting a price oracle

Severity: High
Type: Data Validation
Target: `Comptroller.sol`

Difficulty: Medium
Finding ID: Compound-TOB-026

Description

The tests for ensuring proper reversion when a contract does not implement `isPriceOracle` during invocation of `_setPriceOracle` are skipped, resulting in no test coverage for the negative test case (a contract that does not `implement isPriceOracle`).

```
   it.skip("reverts if passed a contract that doesn't implement isPriceOracle", async () =>
{
      await assert.revert(send(comptroller, '_setPriceOracle', [comptroller._address]));
      assert.equal(await call(comptroller, 'oracle'), oldOracle._address);
    });

    it.skip("reverts if passed a contract that implements isPriceOracle as false", async ()
=> {
      await send(newOracle, 'setIsPriceOracle', [false]); // Note: not yet implemented
      await assert.revert(send(notOracle, '_setPriceOracle', [comptroller._address]),
"revert oracle method isPriceOracle returned false");
      assert.equal(await call(comptroller, 'oracle'), oldOracle._address);
    });
```

Figure 26.1: The skipped tests in `test/Comptroller/comptrollerTest.js`*.*

The requirement of `isPriceOracle` is commented out, resulting in no validation of the `newOracle`.

```
   /**
      * @notice Sets a new price oracle for the comptroller
      * @dev Admin function to set a new price oracle
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setPriceOracle(PriceOracle newOracle) public returns (uint) {
        // Check caller is admin OR currently initialzing as new unitroller implementation
        if (!adminOrInitializing()) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PRICE_ORACLE_OWNER_CHECK);
        }

        // Track the old oracle for the comptroller
        PriceOracle oldOracle = oracle;

        // Ensure invoke newOracle.isPriceOracle() returns true
        // require(newOracle.isPriceOracle(), "oracle method isPriceOracle returned false");

        // Set comptroller's oracle to newOracle
        oracle = newOracle;

        // Emit NewPriceOracle(oldOracle, newOracle)
        emit NewPriceOracle(oldOracle, newOracle);
        return uint(Error.NO_ERROR);
    }
```

Figure 26.2: The `_setPriceOracle` *function definition, displaying the commented out* `isPriceOracle` *requirement.*

Exploit Scenario

A contract that does not adhere to the `PriceOracle` interface is supplied as the `newOracle` parameter. Because `_setPriceOracle` does not properly validate that the supplied oracle is a price oracle beyond interface adherence, the wrong oracle could be set by mistake.

Recommendation

In the short term, enforce the `isPriceOracle` requirement when setting the price oracle.

In the long term, ensure test coverage includes both positive and negative tests of success.

27. Invalid documentation regarding upper-bounds error checks

Severity: Low
Difficulty: Low
Type: Documentation
Finding ID: Compound-TOB-027
Target: `Across the codebase`

Description

Across the contracts, there are several areas which refer to the constant value `-1` in documentation, but, in fact, refer to `uint(-1)`, which is the maximum value of a `uint`. This could result in improper assumptions about functionality and lead to misunderstanding developer intent.

```
       /* Fail if repayAmount = -1 */
        if (repayAmount == uint(-1)) {
            return fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED,
FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX);
        }
```

Figure 27.1: A snippet from `CToken.sol` *highlighting the invalid documentation.*

Exploit Scenario

A developer reading the documentation interprets `-1` as representing a signed integer, but the implementation uses `uint(-1)` as a way to refer to the maximum value of an unsigned integer. This disparity results in misinterpretation of intent, potentially leading the developer to introduce breaking changes.

Recommendation

In the short term, ensure documentation is accurate throughout the codebase.

In the long term, ensure testing reflects the documentation, ensuring correctness of both documentation and implementation.

28. Missing tests and functionality for checking senders when setting allowSeize

Severity: Undetermined
Type: Data Validation
Target: `liquidateTest.js, BoolComptroller.sol`

Difficulty: Low
Finding ID: Compound-TOB-028

Description
Within the liquidateTest.js file, there is a test with a comment, indicating a lack of tests to ensure callers are properly checked. These tests run against the `BoolComptroller` contract attached to a `CToken`, created by several helpers.

```
 before(async () => {
    cToken = await makeCToken({comptrollerOpts: {kind: 'bool'}});
    cTokenCollateral = await makeCToken({comptroller: cToken.comptroller});
  });
...
```

Figure 28.1: The creation of the `CToken` *used in the tests for* `setSeizeAllowed` *unit tests.*

```
async function makeComptroller(opts = {}) {
  const {
    root = await guessRoot(),
    kind = 'unitroller-v1'
  } = opts || {};

  if (kind == 'bool') {
    const Comptroller = getTestContract('BoolComptroller');
    const comptroller = await Comptroller.deploy().send({from: root});
    return comptroller;
  }
...
```

Figure 28.2: A snippet of `test/Utils/Compound.js` *displaying the* `makeComptroller` *helper, and how* `BoolComptroller` *is used when* `kind == bool`*.*

```
 describe('seize', async () => {
    // XXX verify callers are properly checked

    it("fails if seize is not allowed", async () => {
      await send(cToken.comptroller, 'setSeizeAllowed', [false]);
      assert.hasTrollReject(
        await seize(cTokenCollateral, liquidator, borrower, seizeTokens),
        'LIQUIDATE_SEIZE_COMPTROLLER_REJECTION',
        'MATH_ERROR'
      );
    });
```

Figure 28.3: The unit test for ensuring a `setSeizeAllowed` *to false results in a failed* `seize` *call.*

Upon checking the functionality of the BoolComptroller contract, it appears there is no validation of the callers.

```
  function setSeizeAllowed(bool allowSeize_) public {
       allowSeize = allowSeize_;
   }
```

Figure 28.2: The BoolComptroller.setSeizeAllowed *function definition.*

Exploit Scenario

An attacker sets the allowSeize variable to false through the invocation of the setSeizeAllowed function. This results in subsequent seize calls returning an error.

Recommendation

In the short term, ensure callers are appropriately checked for the functions in the BoolComptroller.

In the long term, implement tests to cover the expected callers of the BoolComptroller functions.

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

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

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.

- 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`.

- The `*Verify` functions in the Comptroller include useless logic, enclosed in `if( false ) { ... }` blocks. This is to trick the Solidity compiler into *not* using the `STATICCALL` calling convention for these functions. It relies on the Solidity compiler to optimize out the unreachable code. A new developer inspecting the code might not understand its purpose and delete it, since it is effectively a `NOOP`. At a minimum, the purpose of these blocks should be explicitly stated in code comments.

- EIP 1283 has the potential to save a significant amount of gas with the current re-entrancy guard mechanism. However, 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. For now, the current approach is the best means of re-entrancy protection. Moving forward, we recommend tracking the progress of EIP 1153, which proposes an alternative re-entrancy protection mechanism, with the promise of being more gas-efficient.

- The use of `SafeMath` or `CarefulMath` is missing from the `CToken._reduceReservesFresh` function. Instead, it manually checks for over- and under-flow conditions. Consider refactoring this to use `SafeMath` or `CarefulMath`.

- Expand unit testing to cover the `Exponential.lessThanExp` function. Currently, there are no unit tests covering its expected functionality directly. Existing unit tests only cover it indirectly through usage in other functions.

- The `test/Comptroller/unitrollerTest.js` script contains a skipped test with no body, which should test to ensure the `Unitroller._acceptImplementation` function does not accept a `pendingImplementation` if it is of `address(0)`. Upon further investigation, `Unitroller._acceptImplementation` does check for this case. A unit test should be added to ensure this functionality remains unchanged.