PUBLIC

Code Assessment

# of the cToken Smart Contracts

Nobember 26, 2021

Produced for

Compound

by

CHAINSECURITY

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 8
4 Terminology 9
5 Findings 10
6 Resolved Findings 14
7 Notes 19

1 Executive Summary

Dear Jared,

Thank you for trusting us to help Compound with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of cToken according to Scope to support you in forming an opinion on their security risks.

Compound implements a platform for lending and borrowing. Individual markets on this platform are called cTokens. In principal, there exists one cToken per supported underlying token which allows users to deposit or borrow that underlying token.

The most critical audit subjects are functional correctness, access control, and error handling. We found minor deviations from the functional specification which were reported. With respect to access control, we found that reentrancies allowed access to functionality that was not meant to be executable. Lastly, as the error handling had been revised compared to the previous version, we reviewed it and suggested multiple improvements to clarify it and make it more efficient.

The general audit subjects covered include trustworthiness, documentation, and gas efficiency. Regarding trustworthiness, while cTokens are fully upgradable in order to support future protocol versions without migration, there is a timelock mechanism to allow users to react to changes. We found certain parts of the documentation that could be improved so that other projects can better integrate with the Compound protocol. Lastly, possible improvements to gas efficiency are limited due to the upgradable structure, but minor suggestions were made.

In summary, we find that the codebase provides a high level of security. It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	3
• Specification Changed	2
• Acknowledged	1
Low-Severity Findings	11
• Code Corrected	2
• Specification Changed	3
• Risk Accepted	1
• Acknowledged	5

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the `CToken` contract based on the documentation and additional inquiries. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	September 22 2021	c945f35847d9a16dbc4bb2fbfa0b9e2fdda7f22a	Initial Version
2	October 18 2021	14c0f7cbf04ca1b7ef1110182d89082d2ed9da04	Second Version
3	October 27 2021	4a54ec5c55b66ea67d44b76f3056f0ed7229db8c	Third Version

For the solidity smart contracts, an undefined compiler version `^0.8.6` was chosen. Hence, accidents with different deployments are more likely to occur. The settings regarding optimization are unclear at the time of writing.

2.1.1 *Excluded from scope*

All Compound contracts except for the `CToken`, in particular the `Comptroller`, are outside the scope of this engagement.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section we have added a version icon to each of the findings to increase the readability of the report.

Compound offers money markets for supplying and borrowing different assets on the Ethereum blockchain. Users can supply assets to the market, earning interest on their deposits. They can also use their deposited assets as collateral in order to borrow assets from other markets. The borrowed assets accrue interest over time, which is shared between the suppliers and the protocol. If a borrower's account balance falls below a certain threshold, due to the value of their collateral falling or the value of the borrowed assets increasing, their position can be liquidated. The liquidator pays back the borrowed assets and in return they earn a portion of the borrower's collateral.

2.2.1 *cToken Contracts*

Users interact with the cToken contracts. These are ERC-20 tokens that represent the assets a user has supplied to the market. As the market accrues interest, the value of the cToken compared to the underlying asset increases. The cToken itself receives a portion of the interest as *reserves*. The exchange rate is calculated as

```
(totalCash + totalBorrows - totalReserves) / totalSupply,
```

where `totalSupply` is the total amount of minted cTokens. It is important to note that the initial exchange rate is not necessarily 1 underlying to 1 cToken, in fact in most or all cases a different initial exchange rate was chosen.

The cToken contracts rely heavily on the **Comptroller**. While they provide all the functionality, the Comptroller determines what users are actually allowed to do. All market operations, with the exception of accruing interest, first call the Comptroller to make sure the user has permission to do so. This is also important because certain market operations, e.g. minting new cTokens, can be paused individually in the Comptroller.

All market operations first call `accrueInterest` before any other changes are made, which calculates the interest that must be paid on borrowed tokens. Interest can only be accrued once per block. Note that interest compounds over each accrual, but if multiple blocks have passed without interest being accrued, the interest is calculated **linearly** in the number of blocks and not compounding (exponential).

Users deposit assets into the market using `mint` and receive the appropriate amount of cTokens. They can withdraw their assets using the `redeem` function, which removes the redeemed cTokens from the total supply. If a user has enough collateral (in any market), they can call `borrow` to borrow some tokens. The Comptroller makes sure their account is eligible for borrowing by calculating the sum of their collateral and borrows over all cToken contracts. When a user wants to return their borrowed tokens, they can call `repayBorrow` to pay back their debt. In fact, one can also pay back another user's debt, though this functionality is mostly intended for liquidation. When a user's account liquidity has a shortfall, i.e. their provided collateral no longer covers their borrowed tokens, their position can be liquidated. This can be done by anyone except the borrower themselves. This is done by calling `liquidateBorrow` and providing an amount of tokens to pay back the underwater user's debt. In return, you get a discounted rate on their collateral. The Comptroller decides whether a user's liquidity has a shortfall by summing up their total provided collateral and borrowed tokens. Internally, the cTokens also call a `seize` function to transfer seized collateral from a liquidated user to the liquidator.

As the cTokens are ERC-20 implementations, they conform to the token standard. In particular, the various `transfer` functions also defer to the Comptroller to make sure the user is permitted to transfer tokens, as their account may already have too few funds to cover their borrow balance.

There are also a few additional view functions to aid usability. Note that some functions like `exchangeRateCurrent` trigger interest accrual, so they can actually change the state of the market.

2.2.2 Comptroller

As mentioned above, the Comptroller is the central authority on which market operations are permitted. Each market operation has an `allowed` and a `verify` function. For example, when minting new tokens, first the cToken calls `mintAllowed`. If this check passes, the cToken collects the deposited assets, mints the new cTokens and transfers them to the user. Then, it calls `mintVerify` to pass any post-conditions imposed by the Comptroller. These functions allow the Comptroller both to distribute Comp tokens each time an operation is executed, but also to pause individual functionalities of each market. It also can keep track of all a user's assets and borrowed assets in order to determine whether a user can be liquidated. Additionally, a user's membership to each market is kept track of, so they do not borrow too many different assets.

In `mintAllowed`, the Comptroller simply checks that minting is not paused. `mintVerify` does nothing.

In `redeemAllowed`, the Comptroller checks to make sure the user has enough liquidity to withdraw the specified amount of tokens. `redeemVerify` checks that if no cTokens are redeemed, e.g. due to rounding errors, nothing is actually redeemed. Note that redeeming cannot be paused, so users are able to redeem at any time, provided enough liquidity is available.

`borrowAllowed` first checks that borrowing is not paused. If the user is not already a member of the market they are borrowing from, they are added. Then, it is checked that the user's new borrow balance does not exceed the market's borrow cap. Lastly, the Comptroller calculates the user's account liquidity, making sure they have enough collateral to cover the borrowed tokens. `borrowVerify` does nothing.

`repayBorrowAllowed` does nothing except distribute Comp tokens. `repayBorrowVerify` also does nothing.

`liquidateBorrowAllowed` calculates the borrower's account liquidity to make sure they have a shortfall. Then it checks to make sure the liquidator is not paying off more than `closeFactor` of the borrower's debt. In case a market is deprecated, any account can be liquidated regardless of shortfall in order to move funds to new markets. `liquidateBorrowVerify` does nothing.

`seizeAllowed` only makes sure that both the collateral and borrowed tokens have the same comptroller. `seizeVerify` does nothing.

`transferAllowed` performs the same checks as `redeemAllowed`, as the results are quite similar. `transferVerify` does nothing.

2.2.3 Collateral and Liquidation Details

When a user borrows tokens, they must provide enough collateral so that their account has positive liquidity. Actually, they must provide more than just an equivalent value, as each market has a `collateralFactor`, which indicates the portion of the value of the underlying asset which can be borrowed. The total amount of value a user can borrow (their "borrowing capacity") is determined by the sum of their provided token balances times the respective market's collateral factor. A user cannot perform any action which would cause their borrow balance to exceed their borrowing capacity.

In case the value of a user's borrowed tokens exceeds their borrowing capacity, e.g., because their borrowed tokens increase in value, they are eligible to be *liquidated*. Any other user may pay off a portion of the borrower's debt. The liquidator in return gets a portion of the borrower's collateral at the market rate times the `liquidationIncentive`. The maximum portion of the borrower's debt which can be paid off is determined by the `closeFactor`.

The market rates of the various tokens are defined by price oracles, which are outside of the scope of this audit.

2.2.4 Privileged Roles

Besides the already mentioned roles, the cTokens only have one privileged role:

Admin of cToken: Among other things the admin can upgrade the entire implementation, update important parameters (and thereby also deprecate the market), increase or decrease the reserves, and chose and interest rate model.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. We draw attention to the fact that due to inherent limitations in any software development process and software product an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third party infrastructure which adds further inherent risks as we rely on the correct execution of the included third party technology stack itself. Report readers should also take into account the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severities. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	1

- Reentrancy to Circumvent Liquidation Protection Acknowledged

Low-Severity Findings	6

- Deprecation Insufficiently Documented Acknowledged
- Extra Encoding and Decoding in CErc20Delegator Acknowledged
- Extra Storage Operations Acknowledged
- No Dynamic Bounds on Liquidation Incentive Risk Accepted
- Reentrancy by Admin Acknowledged
- Unnecessary Memory Copies Acknowledged

5.1 Reentrancy to Circumvent Liquidation Protection

Security Medium Version 1 Acknowledged

When a borrow is no longer sufficiently collateralized, it can be liquidated. During a liquidation, the function `liquidateBorrowAllowed` of the `Comptroller` is called to determine if the position can be liquidated. This check basically evaluates whether the values of all held cTokens times their collateralRatio exceed the value of all borrowed assets. The function `liquidateBorrowAllowed` also determines whether the repaid amount does not exceed the `closeFactor`.

However, the following reentrancy attack is possible to circumvent the liquidation protection. We assume that the victim account *V* has two borrowed tokens *A* and *B*. Also, *V* has collateral deposits *C* and *D* and has just become liquidatable, due to a tiny shortfall. We call the respective cTokens *cA*, *cB*, *cC*, and *cD*. Lastly, *A* is contract with a callback, e.g. ERC777.

1. The attacker calls `liquidateBorrow` on *cA* with collateral *cC*. `liquidateBorrowAllowed` is evaluated by the comptroller and determines a small shortfall. Hence, the liquidation is allowed.
2. The token `transferFrom` of *A* is triggered and hence, the callback to the attacker is executed.

    1. As part of the callback, the attacker calls `liquidateBorrow` on *cB* with collateral *cD*. `liquidateBorrowAllowed` is evaluated by the comptroller and determines a small shortfall (as no state changes have yet been performed). Hence, the liquidation is allowed.

   2. The biggest possible amount of *B* tokens is repaid and *cD* tokens are received as reward. The position of *V* is now safe again.

3. Despite the position being safe, the original liquidation continues and the biggest possible amount of *A* tokens is liquidated to received *cC* as reward.

Please note that the attack also works against a single collateral, so if *C* == *D*. It also works with more than two borrowed tokens. In such cases more "parallel" liquidations are possible.

Acknowledged:

Compound has acknowledged the issue.

5.2 Deprecation Insufficiently Documented

Correctness | Low | Version 2 | Acknowledged

The deprecation of a `cToken` and its effects are insufficiently documented. The documentation says:

> *A user who has negative account liquidity is subject to liquidation*

However, liquidation can also occur once a `cToken` has been deprecated. As users aim to avoid liquidation, they should be made aware of this.

Acknowledged:

The Compound team has acknowledged this issue.

5.3 Extra Encoding and Decoding in CErc20Delegator

Design | Low | Version 1 | Acknowledged

The `CErc20Delegator` contract will be used as a proxy. Hence, it generally forwards the calls. However, it contains two ways of forwarding:

1. The generic forwarder using the `fallback` function

2. Explicit forwarders such as:

```
function borrow(uint borrowAmount) override external returns (uint) {
    bytes memory data = delegateToImplementation(abi.encodeWithSignature("borrow(uint256)", borrowAmount));
    return abi.decode(data, (uint));
}
```

The explicit forwarders are less gas efficient as they perform extra decoding and encoding for inputs as well as decoding and encoding for outputs, which is not performed by the generic forwarder.

Acknowledged:

Compound acknowledges the issue but claims that the gas savings are small enough to not be worth fixing.

5.4 Extra Storage Operations

Design Low Version 1 Acknowledged

Inside the function `_acceptAdmin` there is the following code:

```
// Store admin with value pendingAdmin
admin = pendingAdmin;

// Clear the pending value
pendingAdmin = address(0);

emit NewAdmin(oldAdmin, admin);
emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);
```

To emit the events, `admin` and `pendingAdmin` will be queried from storage which is unnecessary here. Hence, there is a certain (even though small due to EIP-2929) gas overhead.

Acknowledged:

Compound acknowledges the small gas savings but deems the readability of the code to be more important.

5.5 No Dynamic Bounds on Liquidation Incentive

Design Low Version 1 Risk Accepted

It is important that the liquidation incentive is sufficiently high in order to provide a safe protocol. However, the product of liquidation incentive and collateral factor also should not exceed `1`. Otherwise, the protocol is sure to lose funds on liquidations.

Risk accepted:

Compound accepts the risk of possibly misconfiguring a protocol.

5.6 Reentrancy by Admin

Security Low Version 1 Acknowledged

In the case of Compound, the `admin` role of the cTokens is held by the governance. Hence, `admin`-based attacks are especially unlikely. However, in principle the admin could perform certain reentrancy-based attacks using special admin functions like `_setComptroller` or `_setInterestModel`. These functions do not have a reentrancy guard.

In a general case, an `admin` could switch out the comptroller for the initial checks of a liquidation and then call `_setComptroller` while receiving a token-based callback. The corrected comptroller address would satisfy the further checks during seizing. This way the `admin` of one market could attack other markets.

Acknowledged:

This issue has been acknowledged.

5.7 Unnecessary Memory Copies

Design Low Version 1 Acknowledged

In some parts of the code there are unnecessary copy operations to and from memory. Consider the following example:

```
function balanceOfUnderlying(address owner) override external returns (uint) {
    Exp memory exchangeRate = Exp({mantissa: exchangeRateCurrent()});
    return mul_ScalarTruncate(exchangeRate, accountTokens[owner]);
}
```

The `exchangeRate` is stored in memory and then directly afterwards copied back onto the stack. However, as the gas overhead of memory operations is tiny, this is minor.

Acknowledged:

Compound has acknowledged the issue but decided not to fix it at this time, as it is only a small gas saving.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	2

- Incorrect Exchange Rates Due to Incorrect Accounting **Specification Changed**
- Incorrect Return Value for mintFresh **Specification Changed**

Low-Severity Findings	5

- Liquidation Incentive Has Imprecise Documentation **Specification Changed**
- Ignored Return Values **Code Corrected**
- Special Case Not Clearly Specified **Specification Changed**
- Unclear Specification **Specification Changed**
- Unnecessary Overflow Checks **Code Corrected**

6.1 Incorrect Exchange Rates Due to Incorrect Accounting

Design **Medium** **Version 2** **Specification Changed**

Due to the design of the `cToken` non-liquidatable borrows can exist. These borrows are incorrectly accounted leading to overly high exchange rates, which can have different consequences. Please consider the following example. For simplicity of the calculation, we assume rates and the liquidation incentive to be zero. We also assume that the exchange rate is 1 cETH = 1 ETH.

1. User *A* deposits 1 ETH, to obtain 1 cETH. At this time 1 ETH is worth 2000 DAI.
2. User *A* borrows 1000 DAI.
3. The price of ETH drops a lot until it is 500 DAI. During this time user *A* is not liquidated (e.g., due to high gas prices).
4. Now user *B* liquidates the DAI-borrow of user *A*.
    - User *B* pays 500 DAI.
    - The amount of seized collateral is computed as 1 cETH.
    - 1 cETH is seized from user *A*.
5. As a result, user *A* now has the following status:
    - 0 balance in cETH
    - 500 borrowed DAI

6. Thereby, user *A* has a non-liquidatable borrow as any liquidation fails in the following line of `seizeInternal` due to an underflow:

```
accountTokens[borrower] = accountTokens[borrower] - seizeTokens;
```

7. This results in an incorrect exchange rate for cDAI. The 500 DAI borrowed by user *A* will never be repaid. However, they are still part of the `totalBorrows` of the accounting within cDAI. Hence, the calculated exchange rate is too large.

The incorrect exchange rate can have different consequences. One example (assuming no reserves) would be:

- All borrowers (except for *A*) are repaying their loans.
- All suppliers try to redeem their deposits. However, each supplier is receiving too much DAI for their cDAI as the exchange rate is too large.
- In the end the last supplier finds that there are 0 DAI inside the contract and still 500 DAI borrowed. As the last borrow is non-liquidatable and will never be paid back, the last supplier cannot redeem their cDAI.

Specification changed:

The Compound team will update the documentation to correctly reflect this behaviour.

6.2 Incorrect Return Value for `mintFresh`

Correctness Medium Version 1 Specification Changed

The `mintFresh` function that is internally responsible of minting new cTokens, has the following specification regarding its return value:

```
* @return (uint) the actual mint amount.
```

At the end of the function, it says:

```
return actualMintAmount;
```

However, the `actualMintAmount` variable contains the amount of underlying tokens used for minting and not the amount of minted cTokens.

Specification changed:

The specification was changed to match the implementation.

6.3 Liquidation Incentive Has Imprecise Documentation

Correctness Low Version 2 Specification Changed

The documentation for the liquidation incentive says:

> *The additional collateral given to liquidators as an incentive to perform liquidation of underwater accounts. For example, if the liquidation incentive is 1.1, liquidators receive an extra 10% of the borrowers collateral for every unit they close.*

However, this does not match the functionality of the code. The function `liquidateCalculateSeizeTokens` will calculate this amount with the liquidation incentive included, but in the function `seizeInternal` the protocol's share is deducted:

```
uint protocolSeizeTokens = mul_(seizeTokens, Exp({mantissa: protocolSeizeShareMantissa}));
uint liquidatorSeizeTokens = seizeTokens - protocolSeizeTokens;
```

Hence, liquidators receive less than 10% extra.

Specification changed:

The Compound team will update the protocol documentation to describe this behaviour more precisely.

6.4 Ignored Return Values

Design | Low | Version 1 | Code Corrected

The return values of "Internal" functions such as `repayBorrowInternal` or `mintInternal` are being ignored in all of their calls. Hence, it could be checked if a return value is really necessary for these functions and if so, whether it should be checked by the callers.

Code corrected:

The unused return values of the relevant functions were removed.

6.5 Special Case Not Clearly Specified

Correctness | Low | Version 1 | Specification Changed

The functions concerning repaying, such as `repayBorrow`, `repayBorrowBehalf`, `repayBorrowBehalfInternal`, `repayBorrowFresh`, and `repayBorrowInternal` generally specify the input variable as:

```
* @param repayAmount The amount to repay
```

However, this variable has a special meaning if it is `-1` as then, the full amount is repaid. This should be documented more clearly in the code.

Specification changed:

The specification was changed for the relevant functions.

6.6 Unclear Specification

Correctness | Low | Version 1 | Specification Changed

The `transferTokens` function is specified as follows:

```
/* ...
 * @return Whether or not the transfer succeeded
 */
function transferTokens(address spender, address src, address dst, uint tokens) internal returns (uint) {
```

However, as the return value is not `boolean` but `uint` it would be beneficial to explicitly state which values indicate success and failure.

The `exitMarket` function in the comptroller has a similar specification:

```
/* ...
 * @return Whether or not the account successfully exited the market
 */
function exitMarket(address cTokenAddress) override external returns (uint) {
```

Again, it would be good to specify which return values indicate success and failure.

Specification changed:

The specification was changed to indicate what happens in case of success and failure.

6.7 Unnecessary Overflow Checks

Design Low Version 1 Code Corrected

In the `doTransferIn` function of the CErc20 contract, the new balance is checked to be larger than the balance before the transfer.

```
require(balanceAfter >= balanceBefore, "TOKEN_TRANSFER_IN_OVERFLOW");
return balanceAfter - balanceBefore;   // underflow already checked above, just subtract
```

However, this check is now unnecessary because of the updated compiler version, which automatically checks for under- / overflows. Therefore, gas can be saved by omitting this check.

Additionally, in the `_addReservesFresh` function, there is still an overflow check.

```
totalReservesNew = totalReserves + actualAddAmount;

/* Revert on overflow */
require(totalReservesNew >= totalReserves, "add reserves unexpected overflow");
```

Because `actualAddAmount` is an unsigned integer, this condition can never occur, as any overflow will be caught by the automatic check by the solidity compiler.

Similarly, the `_reduceReservesFresh` function has an unnecessary check, since the subtraction would revert in case of an underflow.

```
totalReservesNew = totalReserves - reduceAmount;
// We checked reduceAmount <= totalReserves above, so this should never revert.
require(totalReservesNew <= totalReserves, "reduce reserves unexpected underflow");
```

Code corrected:

The unnecessary checks have been removed.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report.

As the scope of this report was limited to the `cToken`, we also list issues outside of the scope in this section.

7.1 Compatibility With Different Tokens

Note **Version 1**

In the future, new tokens might be added. When markets for those are created, issues can appear. In this non-exhaustive list, we highlight some of those issues:

- *On-demand Balance Modification + Callback*:

  Different token types (inflationary, deflationary, or rebasing) can have balances which change without a Transfer occurring. For some of these tokens there is a permissionless trigger to update everyone's balances. Tokens with such a permissionless trigger and a callback on transfer should not be added for the following reason. While receiving the callback of `mint()` the depositor could trigger the balance adjustment and thereby increase the ERC20 balance of the market without making a deposit.

- *Blacklist, Freezable, Seizable*:

  Tokens where some addresses can be blacklisted, certain funds can be frozen or some funds can be seized/burnt, need to be added with great consideration. A blacklisted market would stop working properly. A (partially) frozen market would not function correctly (as the underlying fungibility assumption is violated). Finally, seizing could lead to sudden drops in the exchange rate.

- *Transfer Fees*:

  In principle the protocol supports tokens with transfer fees. However, if a user borrows a certain amount of tokens with transfer fees, it will be almost impossible to completely repay that borrow. This is because the existing feature of providing `-1` as the amount wouldn't work due to the transfer fees. Hence, a small borrow residue will most likely remain.

  When borrowing tokens with transfer fees, the requested amount will not be received. Similarly, when reducing the reserve of a token with transfer fees, there will be unexpected losses.

- *Tokens with potential for sudden increase in value*:

  If a token whose value can suddenly increase by a significant amount, can be borrowed, then attacks due to extremely bad positions are possible. Such tokens include UniswapV2 and Curve pool tokens, but also DPI tokens. Extreme care has to be taken, when adding such tokens to the protocol as they will most likely lead to an attack.

7.2 Hindering Liquidation

Note **Version 1**

Borrowers can do different things to make their own liquidation less likely. During a liquidation the liquidator receives a liquidation reward, determined by the `liquidationIncentiveMantissa` variable. If *X* is the amount of borrowed tokens, the liquidator receives at most:

X * (`liquidationIncentiveMantissa` - 1) * `closeFactorMantissa` / 10**36

Example: If the liquidation incentive is 108% and the close factor is 50%, the maximum liquidation reward is 4% * *X*.

The liquidator needs to pay the transaction costs which will vary over time. At the time of writing the transaction costs for a liquidation are around 80 USD. Hence, the liquidation incentive only provides a sufficient incentive for borrows above 2,000 USD.

As this computation is performed per borrowed token a user might decide to borrow 2,000 USD worth of tokens from ten different tokens, hence borrowing 20,000 USD but making liquidation by a simple liquidator unlikely.

Note that due to partial liquidation, caused by the close factor, such small borrows can also be generated over time.

7.3 Impossible Event Orders

Note **Version 1**

In case that one of the underlying tokens has a callback on token transfer, the `doTransferIn` and `doTransferOut` functions can lead to reentrancies. This can lead to event orders that would not be possible without reentrancies.

7.4 Incomplete Compatibility Check

Note **Version 1**

When adding a new market, the `Comptroller` checks compatibility using:

```
cToken.isCToken(); // Sanity check to make sure its really a CToken
```

However, as `isCToken` returns `true` and to be consistent with the `isComptroller` checks, the return value should also be checked.

7.5 Misplaced Comment

Note **Version 1**

The following comment can be found inside the `seizeInternal` function:

```
/*
 * We calculate the new borrower and liquidator token balances, failing on underflow/overflow:
 *  borrowerTokensNew = accountTokens[borrower] - seizeTokens
 *  liquidatorTokensNew = accountTokens[liquidator] + seizeTokens
 */
```

However, this comment does not refer to the code where it is located but to the code further down. Hence, it could be moved closer to the corresponding code.

7.6 Reentrancy Checks Are Necessary

Note **Version 1**

As part of a recently published post on the Compound Community Forum (https://www.comp.xyz/t/punitive-accounting-for-borrow-and-redeem/2247), it was proposed that punitive accounting may allow to remove the reentrancy checks.

> *With the proposed punitive accounting changes, it may now instead be possible to remove the reentrancy checks altogether. To do that, the community should prove to itself that the protocol is now safe to reentrancy attacks altogether. That can be done as an independent later step, after more thorough evaluation.*

However, this would open up some reentrancy attacks. For example, the following sequence of actions could drain a CToken contract, assuming a token with a callback on `transferFrom`:

1. Provide collateral of some sort, then borrow some funds from the CToken.
2. Call `repayBorrow` to pay back your borrowed funds. Your current borrow balance is stored in `accountBorrowsPrev`.
3. `doTransferIn` is called, which triggers the callback of `transferFrom`. In this callback function, borrow more funds from the contract. This updates your borrow balance.
4. When `doTransferIn` returns, your borrow balance is set to `accountBorrowsPrev - actualRepayAmount`, overwriting the updated borrow balance.

Therefore, omitting the reentrancy checks could lead to vulnerabilities for tokens with callbacks on `transferFrom`.

7.7 Underlying as Immutable Variable

Note **Version 1**

The `CErc20` contracts have the following storage variable:

```
address public underlying;
```

As it is not expected to change, it could become an `immutable` variable to save gas costs during execution. Note that while this change would reduce the execution costs of nearly every `CErc20` invocation by roughly 2200 gas, it also implies that the storage layout would be modified which would require close inspection. Furthermore, it would mean that not all `CErc20` proxies could reference the same implementation contract, as the implementation contracts would contain the specific `underlying` address.

7.8 Unused Comptroller Functions

Note **Version 1**

The following `Comptroller` functions are no longer being used by the cTokens and could hence be removed to reduce the code size and thereby the deployment costs:

- `mintVerify`
- `borrowVerify`
- `repayBorrowVerify`
- `liquidateBorrowVerify`
- `seizeVerify`
- `transferVerify`

7.9 Vote Delegation

Note **Version 1**

Token holders who deposit into a CToken have to be aware that, in the case of governance tokens, they are also giving away their voting rights. On tokens such as COMP or UNI, the governance can pick a delegatee for the voting power which is accumulated inside the CToken contract.

7.10 `closeFactor` Bounds Not Checked

Note **Version 1**

The **Comptroller** contract has a minimum and maximum bound for the value of `closeFactor`.

```
// closeFactorMantissa must be strictly greater than this value
uint internal constant closeFactorMinMantissa = 0.05e18; // 0.05

// closeFactorMantissa must not exceed this value
uint internal constant closeFactorMaxMantissa = 0.9e18; // 0.9
```

However, these bounds are never used. In fact, the `closeFactor` can be set to anything, as its value is not checked at all before setting it.

```
/**
  * @notice Sets the closeFactor used when liquidating borrows
  * @dev Admin function to set closeFactor
  * @param newCloseFactorMantissa New close factor, scaled by 1e18
  * @return uint 0=success, otherwise a failure
  */
function _setCloseFactor(uint newCloseFactorMantissa) external returns (uint) {
    // Check caller is admin
    require(msg.sender == admin, "only admin can set close factor");

    uint oldCloseFactorMantissa = closeFactorMantissa;
    closeFactorMantissa = newCloseFactorMantissa;
    emit NewCloseFactor(oldCloseFactorMantissa, closeFactorMantissa);

    return uint(Error.NO_ERROR);
}
```

7.11 `maxAssets` Not Enforced

Note **Version 1**

According to the documentation, a user should only be able to participate in a limited number of markets, namely no more than `maxAssets`.

```
/**
 * @notice Max number of assets a single account can participate in (borrow or use as collateral)
 */
uint public maxAssets;
```

However, this is not enforced. In fact, the value of `maxAssets` is never set or used. In particular, the function `addToMarketInternal` in the Comptroller blindly adds a user to a new market without checking that the user does not exceed this bound.

```
/**
 * @notice Add the market to the borrower's "assets in" for liquidity calculations
 * @param cToken The market to enter
 * @param borrower The address of the account to modify
 * @return Success indicator for whether the market was entered
 */
function addToMarketInternal(CToken cToken, address borrower) internal returns (Error) {
    Market storage marketToJoin = markets[address(cToken)];

    if (!marketToJoin.isListed) {
        // market is not listed, cannot join
        return Error.MARKET_NOT_LISTED;
    }

    if (marketToJoin.accountMembership[borrower] == true) {
        // already joined
        return Error.NO_ERROR;
    }

    // survived the gauntlet, add to list
    // NOTE: we store these somewhat redundantly as a significant optimization
    //  this avoids having to iterate through the list for the most common use cases
    //  that is, only when we need to perform liquidity checks
    //  and not whenever we want to check if an account is in a particular market
    marketToJoin.accountMembership[borrower] = true;
    accountAssets[borrower].push(cToken);

    emit MarketEntered(cToken, borrower);

    return Error.NO_ERROR;
}
```