Compound Verification Report

Certora Team

September 5, 2018

Disclaimer

These results are from an incomplete analysis by a prototype tool, and are provided free of charge. This report is for internal use by Compound and is not to be redistributed outside of Compound. We hope that this information is useful, but provide no warranty of any kind, express or implied. In no event shall the authors be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

1 Executive Summary

We checked contracts in two preliminary versions of code sent to us by Geoff Hayes on July 14 and July 26, 2018. We did a partial analysis of the July 14 version and a more thorough analysis of the July 26 version. Three contracts where verified: `CarefulMath`, `Exponential`, and `PriceOracle`. We wrote logical specifications of methods in each one of these contracts. We also wrote global invariants for the `PriceOracle` contract. All of these specifications were checked with our verification tool, as well as assertions in the code from the original authors.

We discovered **7** issues, and verified **5** global properties. One of the issues (Issue #1 below) was discovered in the July 14 version and fixed in the July 26 version. All other issues exist in the July 26 version.

2 Scope

We checked all methods except for one. The `setPrices` method in `PriceOracle` was not completely checked since it includes loops and arrays which are not supported in our prototype verification tool. Instead, we verified that `setPrices` met our specifications for one and two accounts, only.

In specifications of methods that return an error code, we only made a distinction between errors and non-errors, and did not specify exact error codes for different erroneous scenarios.

The model we use only partially specifies the blockchain environment: block info (`block`), transaction info (`tx`), and message info (`msg`). For example, we did not check that methods may not accept ether transfer (i.e., that `msg.value` must be 0). We also did not check modifiers (i.e., `payable`), but it is important to note that the code we considered did not have any user-defined modifiers.

Three properties of `setPriceInternal` were checked for 64-bit wide integers instead of the usual 256-bit wide integers, and these properties are marked accordingly in the relevant section.

"Issues" below are not necessarily "bugs" or "problems." In some cases, they represent intended behavior. However, we felt that they are examples of contract behavior that would be worthy of discussion.

3 Issues

Issue 1 There is a scenario in which asset prices and anchor prices can be updated freely, contrary to our understanding of the intended behavior. Suppose that in period 0 (blocks 1-239) some anchor price was set. In every price update during period 0, the local anchor price is set to the requested price, regardless of the swing thresholds. The Money Market is updated with that same price, completely ignoring the anchor.

Furthermore, the above applies also to the first update after period 0 finished. Additionally, the anchor will be updated as well, overriding the original anchor.

In the actual ethereum blockchain, we are long past period 0, so even the first update will set the current period to a number greater than the problematic '0'. But if the protocol is deployed in a 'young' blockchain (and maybe in testing/debugging) this may be an issue.

We found this issue in the July 14 version of the code, and confirmed that it was fixed in the July 26 version.

Issue 2 There is a failed assertion in `PriceOracle`, method `setPriceInternal`. The call to `calculateSwing` in `setPriceInternal` is expected to never fail. However, given an asset A, for which a pending price set by Chad (the account authorized to set the anchor price) to some value C, we can generate an input of `setPriceInternal` that triggers an assertion failure. The input is to set the `asset` argument to A, and `requestedPriceMantissa` to any positive value equal at least $\texttt{MAX\_UINT256}/10^{18} + C$, denoted P.

The `calculateSwing` method multiplies the absolute difference of C and P by the scale factor 10^{18}. Since $P \geq \texttt{MAX\_UINT256}/10^{18} + C$, the absolute difference is $D = |C - P| \geq \texttt{MAX\_UINT256}/10^{18}$. When D is multiplied by 10^{18} (in `getExp`, called from `divExp` at the end of `calculateSwing`), the result is a value equal at least `MAX_UINT256`, generating an overflow that is propagated from `divExp` to `calculateSwing`, resulting in failure of the assertion immediately after `calculateSwing` returns. This is a counterexample to the comment just before that assertion that the only possible error is if `anchorPrice` is 0.

An overflow that should be recorded by the error codes manifests as an assertion failure. The possible damage due to Ethereum semantics of assertions is the loss of all gas given to the transaction.

Issue 3 When the price and anchor price of an asset is in the range $[1, 4]$, the price cannot be updated by the poster, because the minimum change of 1 is greater than the maximum allowed swing of 10%. For an anchor $a \in [1, 4]$, the maximal price and the minimal price are equal a. For example, for $a = 4$ the minimal price is

$$\lfloor \frac{4 * (10^{18} - 10^{17}) + 10^{18}/2}{10^{18}} \rfloor = \lfloor \frac{41}{10} \rfloor = 4$$

and the maximal price is

$$\lfloor \frac{4 * (10^{18} + 10^{17}) + 10^{18}/2}{10^{18}} \rfloor = \lfloor \frac{49}{10} \rfloor = 4$$

Therefore, if the anchor price of an asset A is set to a small value a in the range $[1, 4]$, the only way new prices can be set is for Chad to set the anchor price manually until the update values become large enough to work reasonably.

Issue 4 The swing between the capped price returned by `capToMax` and the anchor relative to which the capped price was computed may be larger than the maximal allowed swing. Let $a = 6$ be the anchor price mantissa, and $p = 5$ the mantissa of the price that we cap. The maximal swing mantissa is 10^{17}. In `capToMax`, the minimal allowed price is

$$\lfloor \frac{a * (10^{18} - 10^{17}) + 10^{18}/2}{10^{18}} \rfloor = \lfloor \frac{59}{10} \rfloor = 5$$

Since $p = 5$, we return 5 as the capped price mantissa.

When we calculate the swing of 5 relative to the anchor price mantissa $a = 6$, we get that it is greater than the maximal swing mantissa:

$$\lfloor \frac{|a - 5| * 10^{18}}{a} \rfloor = \lfloor \frac{10^{18}}{6} \rfloor > 10^{17}$$

The possible adverse effects of this issue are mitigated by the fact that in the only place in the code in which `capToMax` is called, `calculateSwing` is also called and checks the result. However, this means that `capToMax` is not fully self-contained, and must always be used in conjunction with `calculateSwing`. Also, logs where capped prices trigger failures of type `SET_PRICE_MAX_SWING_CHECK` may be confusing.

Issue 5 Chad can freeze the protocol by setting an extreme anchor price. When Chad sets a pending anchor, new prices are not capped. Rather, new prices must be within 10% of the pending anchor price set by Chad. Therefore, if Chad sets a pending anchor for an asset A which is very different from the current trend for prices for A, prices will not be updated at all. Chad may also set a very high pending anchor that will trigger overflow errors in `calculateSwing`.

Issue 6 There are avoidable intermediate overflow errors in `addThenSub` in `CarefulMath`. Suppose $a = 10$, $b =$`MAX_UINT256`, $c = 10$. Then $a + b$ overflows, and the method will return an error. But one may expect to return the arithmetically correct result of $a + b - c$ (equal $b =$`MAX_UINT256`). A similar issue happens in methods such as `Exponential`'s `divScalarByExp` method.

Issue 7 Chad and the price poster may be the same entity. The constructor accepts an argument for the price poster, and sets the sender (constructor caller) as Chad (`anchorAdmin`). The constructor does not check that the argument for the poster is different from the sender.

4 Verified Properties

4.1 Global Invariants

Global invariants are properties that must be satisfied (1) after the contract's constructor is called, and (2) after every public method invocation.

All anchors are set to a period which is between 0 and the current period.

$$\forall x : \texttt{address}. \quad \texttt{anchors}[x].\texttt{period} \leq 1 + \lfloor (\texttt{blockNo}/\texttt{numBlocksPerPeriod}) \rfloor \wedge \texttt{anchors}[x].\texttt{period} \geq 0$$

Once an anchor price is set, it must be positive.

$$\forall x : \texttt{address}. \quad 0 < \texttt{anchors}[x].\texttt{priceMantissa} \vee \texttt{anchors}[x].\texttt{period} = 0$$

Once an anchor price is set, anchor period cannot be 0.

$$\forall x : \texttt{address}. \quad \texttt{anchors}[x].\texttt{priceMantissa} > 0 \implies \texttt{anchors}[x].\texttt{period} > 0$$

Asset prices are within 10% of the anchor prices (Note: This uses integer arithmetic, scaled by 10^{18} and rounds down, so the bounds tend to be a little less than 10% above and a little more than 10% below, with larger discrepancies for smaller numbers.

$$\begin{aligned}&\forall x : \texttt{address}.\\&\texttt{assetPrices}[x].\texttt{mantissa} \in\\&\quad [\lfloor((\texttt{anchors}[x].\texttt{priceMantissa} * (\texttt{expScale} - \texttt{maxSwing.mantissa}) + \texttt{halfExpScale})/\texttt{expScale})\rfloor,\\&\quad\ \lfloor((\texttt{anchors}[x].\texttt{priceMantissa} * (\texttt{expScale} + \texttt{maxSwing.mantissa}) + \texttt{halfExpScale})/\texttt{expScale})\rfloor]\end{aligned}$$

Chad and the price poster must be two different entities.

$$\texttt{anchorAdmin} \neq \texttt{poster}$$

4.2 Method Properties

We checked assertions in all methods and checked for violates of the formal pre-conditions (i.e., assumptions) and post-condidtions (i.e., guarantees) that we specified for each function. For readability, we include only English language descriptions of the specifications.

4.2.1 Assertions

We checked for assertion violations in all methods. All assertions held, except for one in `setPriceInternal` (see issue 2, above).

4.2.2 Function Specifications for Contract `PriceOracle`

`constructor(address _poster):`
Sets Chad (`anchorAdmin`) to sender, sets `poster` to `_poster`, and sets `maxSwing`'s `mantissa` field to `maxSwingMantissa`.

`_setPendingAnchor(address asset, uint newScaledPrice):`
Returns an error if caller is not Chad. If caller is Chad, sets the pending anchor price for `asset` to `newScaledPrice`, and returns a success code.

`getPrice(address asset):`
Returns the mantissa of the current price of the asset: `assetPrices[asset]`

`setPrice(address asset, uint requestedPriceMantissa):`
Assuming that the absolute difference of `requestedPriceMantissa` and the pending anchor for `asset`, multiplied by `expScale` does not overflow, *guarantees* that: (1) if caller is not the price poster, returns an error; (2) if caller is the price poster, then the post conditions of `setPriceInternal` is satisfied. The assumption is necessary because of issue 2 (assertion failure in `setPriceInternal`).

`setPrices(address[] assets, uint[] requestedPriceMantissas):`
At the time of writing this report, our tool does not fully support verification of loops. Therefore, we manually rewrote the function to verify two cases: updating the price of a one or two assets. The rewrite for updating a single asset's

price is equivalent to `setPrice`. For the rewrite for updating two assets' prices, we assumed that updates are done in sequential order: the price of the first asset argument is updated before the second asset argument. Both were verified as correct. The original method receives an array of assets and requested price mantissas and checks that they have equal length, and otherwise returns an error. The original method did not contain any checks of the contents of the arrays, such as checking that different assets are updated, or that the order of appearance of assets in the array does not change the final result.

`setPriceInternal(address asset, uint requestedPriceMantissa):`
Requires the caller to be the price poster. Guarantees:

1. If execution failed (an error code is returned), the state of the contract does not change. (In particular, the state for `asset` does not change.)
2. (Verified for 64-bit wide integers) If there is a pending anchor price for `asset`, the input price `requestedPriceMantissa` must be within the pending anchor bounds defined for `asset` (no capping is allowed).
3. If successfully executed (success code is returned), then there is no pending anchor for `asset`.
4. (Verified for 64-bit wide integers) If there is no pending anchor for `asset` and the current block is not in the same period for which the current anchor price of `asset` is defined, then a new anchor price for `asset` is set as the capped value of `requestedPriceMantissa` relative to the current anchor price and the anchor period of `asset` updates to the current period.
5. If there is no pending anchor for `asset` and the current block is in the same period for which the current anchor price of `asset` is defined, then the current anchor for `asset` (price and period) is not changed.
6. (Verified for 64-bit wide integers) If there is a pending anchor for `asset` and a success code is returned, then the new anchor price for `asset` is the pending anchor and the anchor period of `asset` is updated to the current period.
7. If there is no pending anchor for `asset` and a success code is returned, then the price of `asset` is updated to the capped value of `requestedPriceMantissa` relative to the anchor in the end of this invocation.
8. If there is a pending anchor for `asset` and a success code is returned, then the price of `asset` is updated to `requestedPriceMantissa`. (It is correct, because we already required that `requestedPriceMantissa` to be within the bounds of the pending anchor for a success code to be returned.)
9. If a success code is returned, then the anchor price of `asset` cannot be 0.

`setPriceStorageInternal(address asset, uint _priceMantissa):`
The mantissa of the price of `asset` is updated to `_priceMantissa`.

`calculateSwing(Exp _anchorPrice, Exp _price):`
If there are internal overflows or `_anchorPrice` is 0, returns an error code. Otherwise, if there are no internal overflows and `_anchorPrice` is not 0, returns a success code and the value of

$$\lfloor |\texttt{\_anchorPrice.mantissa} - \texttt{\_price}| * \texttt{expScale} / \texttt{\_anchorPrice.mantissa} \rfloor$$

capToMax(Exp _anchorPrice, Exp _price):

Requires that `_anchorPrice` is greater than 0 (otherwise, will cap to 0 and return a 0, which is unwanted). Guarantees: If there are no overflow errors:

1. If the price is below the minimal threshold (i.e. 10% below `_anchorPrice`), returns `_anchorPrice` minus 10% and cap flag is on.
2. If the price is above the maximal threshold (i.e. 10% above `_anchorPrice`), returns `_anchorPrice` plus 10% and cap flag is on.
3. If the price is in the range of 10% of `_anchorPrice`, returns `_price` and cap flag is off.
4. If returned capped price is `_anchorPrice`, then the input price must be equal to `_anchorPrice`. This property is violated - see issue 3.
5. The returned capped price must be within 10% of `_anchorPrice`. This property is violated - see issue 4.