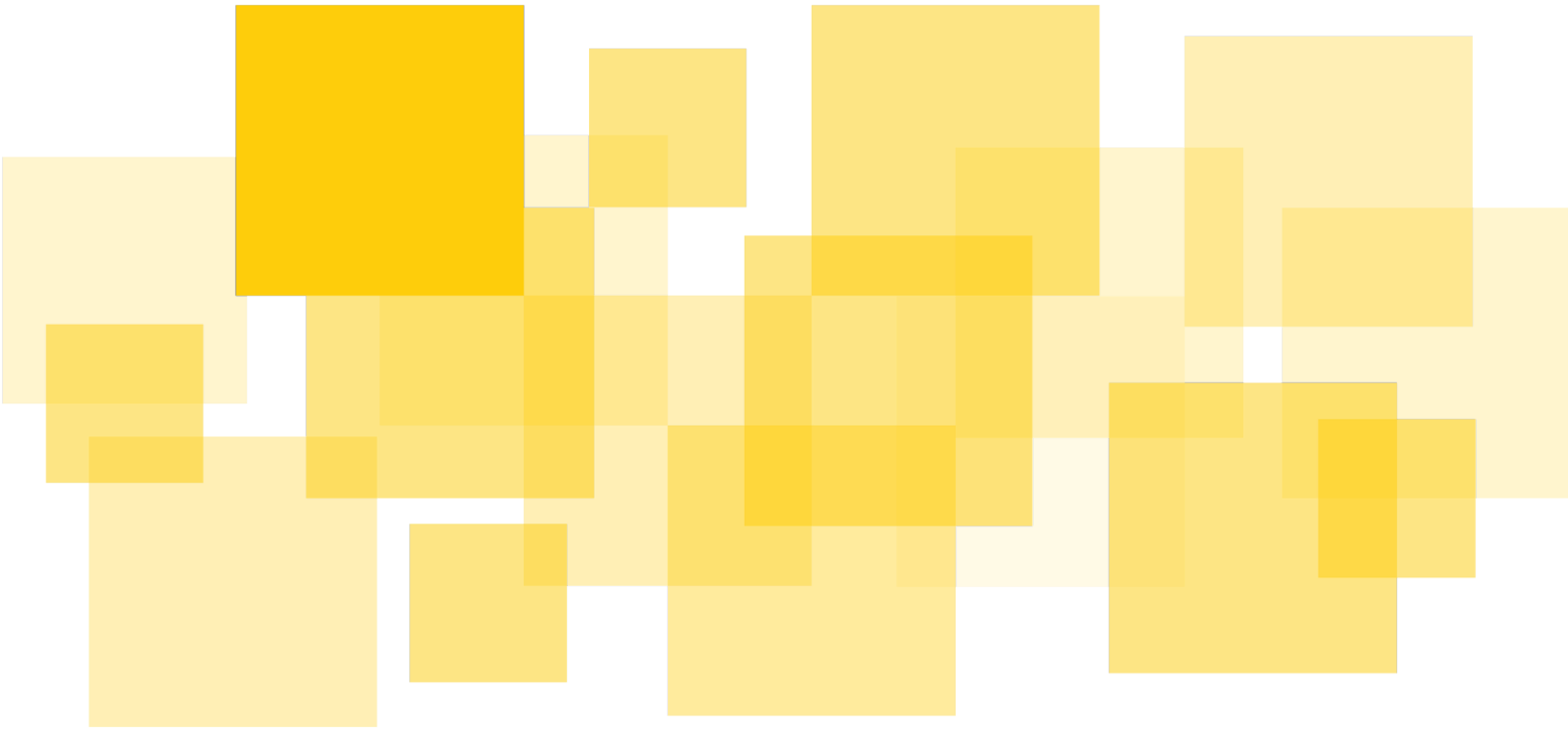
Audit Report

Olympus Pro

Delivered: November 1st, 2021



Prepared for Olympus DAO by Runtime Verification, Inc.

runtime verification

Summary

Disclaimer

Olympus Pro: Contract Description and Invariants

- Olympus DAO controlled: Factory, FactoryStorage and SubsidyRouter
- Partner controlled: CustomTreasury
- Partner controlled: CustomBond
- wOHM

Findings

- A01: No check for correct payout token when creating new bond
  - Scenario
  - Recommendation
  - Status
- A02: Missing checks on bond contracts
  - Scenario
  - Recommendation
  - Status
- A03: Token value calculation
  - Scenario
  - Recommendation
  - Status
- A04: Any user can steal another user's delegated wOHM votes
  - Scenario
  - Recommendation
  - Status
- A05: Reentrancy can bypass debt ceiling and deposit limit with unaffected price

Scenario

Recommendation

Status

A06: The debt ceiling can be manipulated via token supply

Scenario

Recommendation

Status

A07: Price can be manipulated by increased supply

Scenario

Recommendation

Status

A08: The redeem() function uses transfer()

Scenario

Recommendation

Status

A09: Underfunded treasury causes bond price drops

Scenario

Recommendation

Status

A10: Lack of tests for wOHM

Recommendation

Status

A11: Users can temporarily delay other users' redemptions

Scenario

Recommandation

Status

Informative findings

B01: Confusing naming and missing documentation

Status

B02: Unnecessary truncation of uint256

Recommendation

Status

B03: User delegation may block transfers

Scenario

Recommendation

Status

B04: The term “bond” not used in its traditional sense

Recommendation

Status

B05: Anyone can deposit and redeem for anyone

Scenario

Recommandations

Status

B06: Adjustments to the control variable can shoot past the target

Status

B07: Control variable is adjusted after deposit

Status

B08: Bond contracts can be reinitialized

Status

B09: trueBondPrice() is an approximation that overshoots

Recommendation

Status

B10: The truePricePaid is only for the last deposit

Recommendation

Status

B11: Tier ceilings array can be incorrectly constructed

Recommendation

Status

Security properties verified

CustomTreasury

wOHM

Appendix 1: Correctness of getPriorVotes

Appendix 2: Deployment Procedure and Analysis

Deployment

Initial State

Example State

Summary

Runtime Verification, Inc. has audited the smart contracts source code for Olympus Pro. The review was conducted by Rikard Hjort from September 21st to October 22nd, 2021.

The Olympus DAO team engaged Runtime Verification in checking the security of their new Olympus Pro offering, which extends their model of protocol controlled liquidity to include other protocols, called “partners”.

The following issues have been identified:

- Missing checks: A01, A02.
- Potential security vulnerabilities: A04, A05, A06, A07, A08.
- Fault tolerance concerns: A03, A09, A10, A11.

A number of additional suggestions have also been made, namely:

- Usability issues: B01, B03, B04
- Potential integration issues: B05, B09, B10,
- Input validation: B06, B08, B11,
- Best practices: B07
- Gas optimization: B02

The code is mostly well written and thoughtfully designed, following best practices.

Scope

The audited contracts are:

- CustumBond: controls the mechanism by which a protocol sells one token (payout) in exchange for another (principal), at a discount and with a vesting period. For example, it lets a protocol sell its protocol token for a LP tokens with a week-long vesting period. Each protocol may have one or more bonds.
- CustomTreasury: The account which receives the principal tokens from bond sales, and to which the protocol sends the payout tokens so that the bond contract can sell it. Each protocol should have only one treasury.
- Factory: Creates new bonds and treasuries.
- FactoryStorage: Stores relevant mappings between bonds and treasuries for easy querying by other smart contracts.
- SubsidyRouter: Allows governance to set certain addresses “controllers” for bond contracts. These may reset a certain value in a bond contract.

The audit has focused on the above contracts, and has assumed correctness of the libraries they make use of. The libraries are widely used and assumed secure and functionally correct.

In addition to the Olympus Pro contracts, we have also audited the new wrapper contract for staked OHM, called wOHM.

- wOHM: wraps sOHM. sOHM is a rebasing token, whereas wOHM is not, and instead appreciates in value as measured in sOHM; wOHM is redeemable for sOHM, and the redeemable amount increases with time. wOHM also tracks balance snapshots for governance purposes.

The review encompassed two private code repositories, both of which are Hardhat projects with test scripts.

- For Olympus Pro, the code was frozen for review at commit `4acfdfb2f7b2d67ba2bbe858a27dc06fd7c2bd41`.
- For wOHM, the code was frozen at commit `ff229d52d27cbc12713394ba30c7b948a5d31472`.

The review is limited in scope to consider only contract code. Off-chain and client-side portions of the codebase are *not* in the scope of this engagement.

Assumptions

The audit is based on the following assumptions and trust model.

1. OlympusPro requires regular intervention on the part of the partners. We assume the governance address(es) of each CustomBond and CustomTreasury is trustworthy and responsible. They adjust parameters in a sensible way when necessary, and do not mark as a trusted bond contract anything that wasn't deployed as an instance of the audited contract. We assume that the privileged governance address in assumptions will also monitor changes to price, perform token issuances to the CustomTreasury when needed, and perform ongoing due diligence to ensure that the economics of the bond contracts are sound for their intended purpose. It is assumed to be in the self-interest of the governance addresses to protect the protocol rather than damage it, unless they can directly profit from such damage. They safely manage the keys to governance addresses, or to the threshold multisigners, or set up a sensible owner smart contract.
2. The Olympus DAO address likewise acts responsibly, and sets up correct addresses for governance and fee reception. Here too we must assume that the keys are in safe hands and multisigs well protected.
3. All libraries and contracts other than the ones reviewed are secure and bug-free. The interfaces are correct and up to standard. External contracts (such as token contracts) can have bugs, but it falls under assumption 1 and 2 and that token contracts are reasonably vetted.

4. The Solidity compiler introduces no unknown bugs; it preserves the intended semantics. The audit is over the Solidity code, and does not employ EVM inspection techniques. The contracts use Solidity v0.7.5, and none of the reported bug fixes since then in the Solidity release notes affect the contract code reviewed.

Note that assumptions 1-2 roughly assume honesty and competence. However, we will rely less on competence, and point out wherever possible how the contracts could better ensure that unintended mistakes cannot happen.

We do not assume full trust-worthiness of the governance addresses, and do not for example trust with the ability to block certain depositors, prevent redemptions, or take control of funds that the user did not intend to allow (roughly: theft).

The governance addresses may profit in indirect ways by disrupting the protocol, such as by taking large shorts positions against their own tokens, but such considerations lie outside of the scope of the security review, and will only be considered on a best-effort basis.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to known security issues and attack vectors. Thirdly, we discussed the most catastrophic outcomes with the team, and reasoned backwards from their places in the code to ensure that they are not reachable in any unintended way. Finally, we regularly participated in meetings with the Olympus team and offered our feedback during ongoing design discussions, and suggested development practices as well as design improvements.

This report describes the **intended** behavior and invariants of the contracts under review, and then outlines issues we have found, both in the intended behavior and in the ways the code differ from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter. Finally, we also an give an overview of the important security properties we proved during the course of the review.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Olympus Pro: Contract Description and Invariants

This section describes the contracts at a high-level, and which invariants of its state we expect it to always respect at the end of a contract interaction.

Olympus DAO controlled: Factory, FactoryStorage and SubsidyRouter

These contracts are controlled by Olympus DAO (though their ownership address can change). They are currently deployed at the following addresses:

- Factory: 0xb1F69deCb09D8490E3872FE26D27a7b83493cd65.
- FactoryStorage: 0x6828D71014D797533C3b49B6990Ca1781656B71f
- SubsidyRouter: 0x97Fac4EA361338EaB5c89792eE196DA8712C9a4a

The Factory is used to deploy the partner controlled Olympus Pro contracts: CustomBond and CustomTreasury. Both creations can only be performed by the contract owner. The Factory can 1) create a CustomTreasury and a CustomBond in tandem, and 2) create a new bond which connects to an existing treasury. In either case, the bond information is stored in the FactoryStorage, which contains a bi-directional lookup from address to bond information. This lets other contracts enumerate all existing bonds. The FactoryStorage stores all the immutable public information of the bond, along with the "initial owner" ("policy") of the bond.

The subsidy router has no direct influence on the other Olympus Pro contracts. It is a utility for other contracts (not audited here) called "subsidy controllers", and allows them to lookup and interact with the bonds for which they control subsidy. The owner of the subsidy router contract controls the routing, and can route each controller to a single bond. The controller can perform only one function: it can retrieve the amount of payout tokens that the bond has sold since the function last was called. Whenever the function is called, the number is returned and the counter is reset. **The actual subsidy payments are not part of the bond mechanisms, and the bond contracts should never be sent any tokens, except via the treasury.** Keeping track of "payouts since last subsidy" is just a utility of the bond contract, and does not mean it has any ability to handle any token transactions other than exchanging principal tokens for payout tokens.

Partner controlled: CustomTreasury

The treasury controls all the principal tokens and payout tokens of all the bonds of a specific partner. Each treasury may serve several bond contracts. The bond contracts should all use the same payout token, but may have different principal tokens. The owner of the contract sets up the relationship by creating the bond contract with its treasury address (immutable), and by setting a mapping in the treasury detailing which contracts are connected bond contracts, giving them unlimited access to the funds in the treasury. The owner of the contract also has unlimited access, and can withdraw any funds.

The treasury supports only ERC20 funds. It is not capable of transferring other kinds of tokens, or ether. There are also certain restrictions on which tokens should be served, see A03, A05, A06 and A07.

Partner controlled: CustomBond

The bond contracts are the central contracts of the Olympus Pro protocol, and the most complex. They have two user-facing functions: *deposit()* and *redeem()*.

A bond contract is set up to offer a delayed swap between two tokens at a moving price. The tokens addresses are immutable -- they are set at the creation of the bond contract and cannot be changed (though note that the actual token implementations can change, if they are upgradeable). The token which users must deposit is called the "principal" token, and in exchange they receive the "payout" token. The payout tokens vest over a predetermined time period (at least 36 hours, though see B08), and any vested amount is immediately redeemable. The payout P received for a user who pays a given $\_amount$[1] of principal tokens is set as follows:

$$P = \frac{value}{bondPrice()} \cdot 10^7 \text{ where}$$

$$value = \_amount \cdot 10^{payoutToken.decimals - principalToken.decimals} \text{ and}$$

$$bondPrice() = max\left(minimumPrice, \frac{controlVariable \cdot debtRatio()}{10^{payoutToken.decimals - 5}}\right) \text{ and}$$

$$debtRatio() = \frac{currentDebt()}{payoutToken.totalSupply()} \cdot 10^{payoutToken.decimals}$$

This gives us:

[1] Given in the raw amount of the principal token, i.e. ignoring decimals. So for example, a user paying 1 DAI would be giving the $\_amount$ $1 * 10^{18}$, or 1,000,000,000,000,000,000.

$$P = \frac{\_amount \cdot 10^{payoutToken.decimals - principalToken.decimals}}{max\left(minimumPrice, \frac{controlVariable \cdot \frac{currentDebt()}{payoutToken.totalSupply()} \cdot 10^{payoutToken.decimals}}{10^{payoutToken.decimals - 5}}\right)} \cdot 10^{7}$$

Simplified:

$$P = \frac{\_amount \cdot 10^{payoutToken.decimals - principalToken.decimals}}{max\left(minimumPrice, controlVariable \cdot \frac{currentDebt()}{payoutToken.totalSupply()} \cdot 10^{5}\right)} \cdot 10^{7}$$

As soon as the price organically moves over the minimumPrice, the minimumPrice gets set to 0, and the price floor is then decided only by the current market state. For most of the bond contract's life, we expect the minimumPrice will be 0. This leads to further simplifying the price:

$$P' = 100 \cdot \_amount \cdot \frac{payoutToken.totalSupply()}{controlVariable \cdot currentDebt()} \cdot \frac{10^{payoutToken.decimals}}{10^{principalToken.decimals}}$$

The only variable in the equations directly under the control of the user is *_amount*. The only variable in the equations directly under the control of the owners of the contract is *controlVariable*[2]. The *currentDebt* value is the number of unvested[3] payout tokens. The *currentDebt* value adjusts every block, and its speed of adjustment is inversely proportional to the *vestingTerm*, which is also set by the owners of the contract.

The above amounts are the amount of payout tokens paid out by the bond contract for a given deposit of principal tokens. However, not all goes to the buyer. A certain percentage is paid as a fee to the Olympus DAO. The fee is set based on "tiers", which are set at contract creation and immutable. The different fee tiers are defined by ceilings of absolute amounts of bonded principal tokens. Once a ceiling is broken, the fee moves to the next tier. The total amount only increases, and therefore once the contract moves past a tier, it will only ever move to later ones.

The true payout a user receives is:

[2] Note that the control variable works as a control of bond emission velocity. Setting a higher value of the control variable means that the price of bonds increase faster which every purchase, limiting the emission speed (since there is no incentive to ever buy bonds when the price relative to the principal token exceeds the market price elsewhere). Setting a low value on the control variable gives a higher emission of payout tokens, since higher volumes of bond prices are required to bring the price up to or near the market price of the payout token.

[3] Tokens vest at a constant rate per block, so *currentDebt()* reduces every block if there are no new deposits (bond purchases).

$P'' = P \cdot (1 - fee)$ where

$fee = currentOlympusFee() \cdot 10^{-6}$ which gives

$P'' = P \cdot (1 - \frac{currentOlympusFee()}{10^6})$

Note that the fee tiers are given in 100ths of basis points, i.e. at a resolution of 10^6.

The bond contract stores a value in the bond representing the true bond price of the latest bond the user purchased. This is only for information purposes and not used in the protocol.[4]

Invariant: The amount of payout tokens controlled by the bond >= (the amount of payouts returned by deposit - amounts returned from redeem). If no tokens were transferred to the contract independently (by a user, on accident), then the >= becomes ==

Invariant: If no tokens were transferred to the contract independently (by a user, on accident), then the amount of tokens other than the payout token, including principal tokens, is always 0.

Invariant: totalDebt <= maxDebt + maxPayout

Invariant[5]: $sum(redeem(amount)) \leq sum(deposit(amount))$

wOHM

OHM is a regular ERC20 token. By staking OHM, one can obtain an equivalent amount of sOHM. sOHM is a rebasing version of the token, meaning that every user's balance regularly is updated. In the case of sOHM, the rebase always means the balances increase, to reflect the staking rewards. This is done by sOHM having an ever-increasing, contract-wide "index".

wOHM is a wrapper around sOHM. Instead of rebasing, the token balance of a user remains constant unless they send or receive tokens to their address. wOHM is obtained by sending sOHM to the wOHM contract. The amount of wOHM received in exchange is determined by the current *index*. To be precise, the amount of wOHM received for wrapping an amount $\_amount_s$ of sOHM is given by the following formula:

$$W = \_amount_s \cdot \frac{10^{18}}{sOHM.index()}$$

Conversely, the amount of sOHM received by unwrapping an amount $\_amount_w$ of wOHM is given by:

[4] The "true bond price" stored is actually *bondPrice()*(1+fee)*. This is not the exact true price paid, which would be *bondPrice()/(1 - fee)*. However *1+x* is a decent approximation for *1/(1-x)* for small *x*.
[5] If the Olympus fee is non-zero, the sides of the inequality are never equal.

$$S = \_amount_w \cdot \frac{sOHM.index()}{10^{18}}$$

We can see that the prices are simple inverses of each other. If a user were to unwrap an amount and immediately wrap the received sOHM, or vice versa, their balance would remain unchanged.

Invariant: A user calling *wrap(unwrap(amount))* leaves all account balances of both sOHM and wOHM unchanged.

The wOHM contract also keeps track of historical voting balances of wOHM for each holder account. At each transfer or delegate event, the current block number and new voting power of the affected accounts are stored in an array of "checkpoints" for the affected accounts. The view function *getPriorVotes* can be used to obtain the number of wOHM tokens each account held at a specific block (through a binary search on the list of checkpoints for that specific user), which can be used for snapshot votes. The balances are maintained in one array per account.

Holders of wOHM can delegate their voting rights. They can either delegate the rights to themselves, or to another user. By default, they are delegated to the 0x0 address, which is treated by the vote-tracking functions as conferring no voting rights. Users can only delegate their full balance, and only delegate to one address at a time. Whenever their balance changes, through a transfer, a wrap or an unwrap, the number of votes delegated to their delegate also changes by the same amount.

Invariant: for any account A, the list of checkpoints *checkpoints[A]* is sorted according to the block number of the checkpoint in ascending order, and each block number appears at most once.

Invariant: *getCurrentVotes(user)* is equal to the balance of all addresses *x* for which *delegator[x]* is *user*.

Findings

A01: No check for correct payout token when creating new bond

[Severity: Medium | Difficulty: N/A | Category: Input Validation]

When creating a new bond and treasury together, through the *createBondAndTreasury()* in the factory, a single payout token address is used for both. However, for subsequent bond creations using the *createBond()* function, there is no check that the payout token for the bond matches that of the treasury. Since creating bonds is a privileged function to the Olympus DAO, we can assume that competency should prevent incorrect deployments. However, such a deployment could have serious consequences.

Scenario

Olympus launches a bond contract with the incorrect payout token, X. The treasury uses payout token Y. (Perhaps X is a wrapped version of token Y, and the deployers made an understandable mistake.) A user buys bonds from the newly created CustomBond contract, depositing its principal tokens. The bond deposits the principal tokens in the treasury bonds, and receives token Y in return. When the user goes to redeem, the bond contract holds no X tokens, only Y tokens. The Y tokens are then lost forever. This issue only manifests with failures when users go to redeem, and thus will only be detected by contract inspection.

Furthermore, the bond price will have been determined based on the total supply of token X, not token Y, so prices may not be as expected.

In this scenario, if the partner controls the Y token, they can make users whole by minting new payouts in token Y. They may also repay the principal tokens. However, the Y tokens in the bond contract are still lost.

Recommendation

Don't pass the *_payoutToken* parameter to *createBond()*. Instead, read the payout token from the treasury address.

Status

Addressed in commit [863974c1159ef12273598432db6c909366369b17](863974c1159ef12273598432db6c909366369b17).

A02: Missing checks on bond contracts

[Severity: High | Difficulty: N/A | Category: Security]

Any address considered marked as a "bond" in a treasury is whitelisted to withdraw any amount of payout tokens from the treasury. Any bond contract must therefore be thoroughly audited, fully trusted and should not be upgradeable. This is because the *deposit()* function does not perform any price checks, but instead just accepts any prices the caller sets, including accepting 0 payout tokens for any amount of payout tokens.

Scenario

An upgradeable or insecure bond contract is whitelisted in the treasury. If the contract is upgraded or exploited to request too much payout tokens from the treasury, the treasury will accept the payout.

Recommendation

From a security perspective, we would recommend having the treasury perform a minimum price check which the policy address controls, or check that the price hasn't slipped an unreasonable amount since the last deposit. However, the responsibilities of the policy address, the complexity of the solution or its gas costs may make this approach unfeasible. A simpler remedy, which can at least prevent mistaken double toggles or replay attacks is to replace the *toggleBondContract()* function with two idempotent functions, *whitelistBondContract()* and *dewhitelistBondContract()*.

This issue makes the "toggle" function somewhat dangerous, and we'd prefer to see idempotent functions that turn a trusted bond address on and off, to avoid any accidental double toggling if a bond contract or address that turns out to be compromised.

Status

Addressed in commit 04500fb10d563231fa2bc3e85c7a65a4f2ceeb2b.

A03: Token value calculation

[Severity: Medium | Difficulty: N/A | Category: Functional Correctness]

The function *valueOfToken()* in CustomTreasury is to be used with care. In the case of a payout token with far fewer decimals than the principal token, the precision may become low. This means that any project using a payout token with few decimals, principal tokens with many decimals should be avoided.

Scenario

For example, if the payout token has 8 decimals and the principal token 18 decimals, the incoming amount is scaled down by a factor of 10^{10}. Since decimals don't have any intrinsic bearing on price, this can cause significant round-off if the incoming amount is relatively small.

$$valueOfToken(prinicpalToke,\ \_amount) = \_amount \cdot \frac{payoutToken.decimals}{principalToken.decimals}$$

Recommendation

There is no reason to scale values by token decimals specifically, and it would be equally sensible to allow the value scaling to be set by the policy address. Refer to the equations in Section "Partner controlled: CustomBond" to see how this can be tuned.

Status

Acknowledged.

A04: Any user can steal another user's delegated wOHM votes

[Severity: High | Difficulty: Low | Category: Security]

The way bookkeeping of and writing of delegations is performed allows users to steal delegated votes from other users. The issue is that currently, *delegate* transactions move the full balance of the delegator into voting rights for the delegate, while *mint*, *burn*, and *transfer* transactions move delegations from the sender to recipient. This seems to be an error in the refactoring, and is not present in the COMP token which wOHM is based on.

Scenario

User Alice has balance 10, delegated to herself. She delegates to Bob, who has a previous delegation of 10. This gives him a total voting power of 20, stored in his checkpoint M, and giving Alice 0 power, stored in her checkpoint N.[6]

She now wraps some sOHM into wOHM, and is minted 5 wOHM[7]. This triggers *_beforeTokenTransfer*, and so Alice is now delegated a power of 5 from the 0 address, stored in her checkpoint N + 1.

```
    function _beforeTokenTransfer(address from, address to, uint256 amount)
internal {
        _moveDelegates(from, to, amount);
    }
```

Alice now delegates to herself. The *delegate* function calls *_delegate(Alice, Alice)*.

```
   function _delegate(address delegator /*Alice*/, address delegatee
/*Alice*/) internal {
        address currentDelegate = delegates[delegator]; // Bob
        uint delegatorBalance = _balances[delegator]; // 15
        delegates[delegator] = delegatee; // Alice

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }
```

[6] We assume Alice has N-1 checkpoints and Bob has M-1 checkpoints, both non-zero, for simplicity. Alice starts with 10 votes, and Bob with 10.
[7] Alternatively, she is sent 5 wOHM from another address.

```
    function _moveDelegates(address srcRep /*Bob*/, address dstRep
/*Alice*/, uint amount /*15*/) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                uint srcRepNum = numCheckpoints[srcRep]; // M + 1
                uint srcRepOld = srcRepNum > 0 ?
checkpoints[srcRep][srcRepNum - 1].votes : 0; // 20
                uint srcRepNew = srcRepOld.sub(amount); // 5
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew)
// Bob, M, 20, 5
            }

            if (dstRep != address(0)) {
                uint dstRepNum = numCheckpoints[dstRep]; // N + 2
                uint dstRepOld = dstRepNum > 0 ?
checkpoints[dstRep][dstRepNum - 1].votes : 0; // 5
                uint dstRepNew = dstRepOld.add(amount); // 20
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
// Alice, N, 5, 20
            }
        }
    }
```

Recommendation

Change *_beforeTokenTransfer* to the following:

```
    function _beforeTokenTransfer(address from, address to, uint256 amount)
internal {
        _moveDelegates(delegates[from], delegates[to], amount);
    }
```

Status

Addressed in commit [45f5fa531ea1a9532d93321a3f491784b39c314e](45f5fa531ea1a9532d93321a3f491784b39c314e).

A05: Reentrancy can bypass debt ceiling and deposit limit with unaffected price

[Severity: High | Difficulty: Medium | Category: Security]

The intended meaning of the function *maxPayout()* is that it controls how much can be bonded in a single transaction. This guarantees that the price rises stepwise, since the price is recalculated for each bond purchase, and based on the current unvested bonds. Whenever a user buys a bond, they should only be able to buy up to the amount given by *maxPayout()*, and then the price is updated.

Some ERC20 tokens may contain callback functionality in their transfer functions. For example, ERC777 tokens always perform a callback to the sender.

In the *deposit()* function the transfer of principal tokens happens after state variable and price **checks** but before any state variables are **updated**. This means that accepting any principal token which allows callbacks (or which could add that functionality in the future) means that the *deposit()* function could be called repeatedly through reentrancy, and that neither the price nor the total debt bookkept would be updated. This would effectively mean that the debt ceiling becomes pointless to prevent a motivated buyer, who could buy any amount of bonds at the current price. Even if the term variable *maxDebt* is lowered, or even set to 0, the attacker can bypass it as soon as the current debt reaches that level, which would also make the scenario more dangerous, since the lower debt would bring the price down before the attacker strikes. The only way to completely stop payouts is to set a *maxPayout* of 0.

Scenario

The principal token in a new OlympusPro bond performs a callback to the sender. For example, it might be an ERC777 token which calls *tokensToSend()* on the sender (or its delegate) before a transfer. The current price of bonds has fallen to 90% of the market value for payout tokens. Alice is a whale and a motivated buyer who believes that this price will remain the same over the vesting period. She sets up a contract and sends her principal tokens to it. The contract implements a function to call *deposit()* on the bond contract with fewer principal tokens than what would cross the *maxPayout* threshold. It also implements a non-reverting *tokensToSend()* which performs a new call to *deposit()* for as long as there are principal tokens left or the treasury stays funded, i.e. until calling *deposit()* fails. Alice triggers the depositing function in her contract, causing her contract to buy bonds for the whole balance of principal tokens she has available at the 90% price, pushing the actual outstanding debt far past the debt ceiling, bringing the price of bonds far past 100% of market value, and letting her acquire far more tokens than was intended by the partners, at too low a price.

Recommendation

Perform the transfer of tokens in the bond's *deposit()* function **after** the state updates, right before the function returns.

Status

Addressed in commit 77d72bc7a5217db2e5901aabd4203385e4df685e.

A06: The debt ceiling can be manipulated via token supply

[Severity: Medium | Difficulty: High | Category:Functional Correctness]

The debt ceiling is not *maxDebt*, but *maxDebt + maxPayout()*. (This is why the attack in A05 works even if *maxDebt* is set to 0). This is unintuitive, and important for the partners to remember as they set their parameters.

The *maxPayout()* value can increase rapidly in the case of a supply increase in the payout token, for example after a large mint of new tokens to be sent to the treasury.

Some tokens have a supply that can be temporarily increased, which makes these economics susceptible to flash loans.

Scenario

A new DeFi protocol decides to use OlympusPro for incentivizing liquidity providers. They do not have a liquidity mining program to begin with, so their token supply is small and the amount of tokens they emit through OlympusPro is high compared to their total supply. The protocol is DAO controlled, and so prefers big payment installments voted on by the community to the treasury. They fund the CustomTreasury for what they expect will be the next six months. This causes a tripling of their token supply, causing the *maxPayout()* value to triple as well, leading to a large increase in their debt ceiling.

Recommendation

Update *totalDebt* after calculating the payout, and only then perform the check that the *totalDebt* is below *maxDebt*.

Status

Addressed in commit 8ea66d22881bfabc5f8caadc57adaf6de3311da7.

A07: Price can be manipulated by increased supply

[Severity: High | Difficulty: Medium | Category: Security]

The economics of the bonds are closely tied to the total supply of the payout token, in that the price of bonds are inversely proportional to the total supply. In A06 we pointed out how the debt ceiling can be affected by the total supply. However, as seen in the equations in Partner controlled: CustomBond, the price P is proportional to total supply. Any sudden increase in the total supply of tokens (for example flash mints, mints to new investors, a yield farming program, etc) will cause a sudden price drop.

Scenario

A tokens uses a CDP-backed currency as their payout token[8]. Alice takes out a big flash loan (or several), mints a large amount of tokens, then buys bonds in a regular way, and then pays back the debt immediately. This way she can increase the total supply temporarily, dropping the price of the bonds.

Recommendation

Tokens with user-manipulable supplies should not be used as payout tokens in an OlympusPro partner system.

We also advise that any partner using OlympusPro avoid such sudden increases in their total supply, assuming the token is under their control. For example, instead of minting and sending large amounts of the payout token to a DAO treasury to be trickled out over time, the minting should happen in unison with the trickle, by minting a little at a time. This also means that it is ill-advised to mint a large share of payout tokens all at once even for use in OlympusPro, e.g. minting the expected total payout for a year of bonds and sending to the CustomTreasury, as this mint directly drops the price.

Status

Acknowledged.

[8] Or the token can be minted some other way, e.g. flash minted, or through wrapping or exchanging for another token.

A08: The *redeem()* function uses *transfer()*

[Severity: Medium | Difficulty: N/A | Category: Functional Correctness]

The *redeem()* function in CustomBond uses *transfer()* for payout tokens rather than *safeTransfer()* and does not check its return value. This is generally safe, since it is guaranteed that the CustomBond holds a sufficient amount of payout tokens, and the payout token is trusted by the partner. But in a scenario where a transfer might fail and return a boolean "false" instead of reverting, the redeemer does not receive their vested tokens, but their vested amount is updated. The tokens get stuck in the bond contract.

Scenario

It is possible that a bug in the token contract, for example, or a freezing of funds (such as allowed by USDC and USDT) could cause a transfer to fail. The user loses their right to redeem those tokens, but does not receive them.

Recommendation

Use *safeTransfer()*.

Status

Addressed in d396ef932fc41f2706bc1934b0ee234b9dd7ff3a.

A09: Underfunded treasury causes bond price drops

[Severity: Medium | Difficulty: Medium | Category: Security]

If the treasury is underfunded on payout tokens, more bonds cannot be purchased. When bonds cannot be purchased, the outstanding debt gradually falls and with it, the bond price. The bonds are economically designed to always be available, and market mechanisms ensure that the price is kept close to the actual price of the payout token denominated in the principal token. The flow of tokens in the bond purchase transactions are always: principal tokens go in (and are sent to the treasury), payout tokens go out (from the treasury), all in one transaction.

If, for political reasons, high demand, price movements, or any other reason, the treasury ever becomes underfunded and doesn't have enough payout tokens to support new bonds, then new bonds can't be created, but the total debt still falls, reducing the price of bonds. If the treasury is ever funded again, allowing deposits, then bonds can be bought at far below the market value.

As a last resort, the partner may stop bond issuance for a while by setting *maxPayout* to 0, wait for all tokens to be vested, and reinitialize their bonds with new minimum prices.

Scenario

A partner has several bond contracts, all with 5 days vesting period, and a planned issuance schedule. The treasury is funded weekly from a DAO vote. One week, on Monday, the prices of two of the principal tokens drop by around 50%, causing an increased demand for those bonds. The payout tokens issued for the week cannot meet the demand of all the bonds, and the DAO cannot get a vote together fast enough to add more funds to the treasury, or increase the control variable to decrease the speed of bond issuance. As a result, no one can buy bonds for two whole days. The DAO makes control variable adjustments and adds new tokens to the treasury. Now all bonds have a more than 40% discount. The control variable adjusts in increments, meaning the initial bond buyers, putting in large orders, get a big discount on their bonds. The control variable can also adjust only 3% per block, meaning that it cannot be relied on to quickly adjust prices in this situation.

If the reentrancy vulnerability in A05 is present, then the buyer could bypass the debt ceiling and max payouts and buy up all the treasury funds in a single transaction at a big discount, without any price adjustments. To avoid this, the DAO is forced to set *maxPayout* to 0 for a week and reinitialize their bonds once all tokens have been vested.

Recommendation

Partners need to be diligent, and set conservative control variable values to ensure that their treasury is always well funded.

Status

Acknowledged.

A10: Lack of tests for wOHM

[Severity: N/A | Difficulty: N/A | Category: N/A]

There are no tests for the wOHM contract. While it is a fork of well-tested contracts, we would still like to see a test suite for wOHM, especially to test the *wrap* and *unwrap* functions. A simple test of the delegation functions would also likely have caught issue A04 before the security audit.

Recommendation

Write a set of tests for wOHM, testing the invariants given in the description.

Status

Acknowledged.

A11: Users can temporarily delay other users' redemptions

[Severity: Medium | Difficulty: Medium | Category: Functional Correctness]

Any user can deposit for any other user. However, whenever there is a deposit, the vesting period resets. A user or protocol depending on its ability to redeem their bonds at a given date can be sabotaged by a malicious user. This comes at a cost to the attacker, who must buy the minimum allowed amount of bonds (which is hardcoded) for the attacked address. However, if there are serious consequences to not being able to redeem on time the attacker may find it worth their while.

Scenario

AlicePro is a DeFi protocol that integrates with OlympusPro, depositing user balances in OlympusPro bonds. AlicePro has made a large deposit that is currently vesting. Bob runs a competitor to AlicePro, and he knows that AlicePro's contracts have their own built-in timing, and that their users expect to be able to get their funds back on time. AlicePro is implemented to have a reliable deposit schedule, always redeeming right before a new deposit. Bob realises that he can cause harm to AlicePro, shorting their tokens and launching a Twitter campaign against it. Right before AlicePro is set to make a deposit, Bob deposits the minimum amount of principal tokens necessary to AlicePro. The value of his deposit is miniscule compared to the funds AlicePro has deposited. When AlicePro redeems, the protocol only receives a small amount of vested tokens, outraging its users. Now AlicePro's vesting period starts over. Bob can continue this attack indefinitely.

AlicePro can prevent this by claiming her vested tokens more often. Bob can only lock up Alice's vested but unclaimed tokens, and only delay Alice's ability to redeem. This is cheap for Bob to do once or twice, but may not be possible indefinitely.

Recommandation

Users and protocols buying bonds from OlympusPro must be careful not to rely too heavily on timely redemptions. It would be possible for OlympusPro to address the issue by allowing each user more than one simultaneous bond, but that causes higher gas prices and might open up other issues as well as increasing complexity.

Status

Acknowledged.

Informative findings

B01: Confusing naming and missing documentation

[Severity: N/A | Difficulty: N/A | Category: Documentation]

The subsidy router has one function which has an effect on the OlympusPro bond contracts, namely *getSubsidyInfo()*. It returns the current value of the *payoutSinceLastSubsidy* variable of the bond contract, and resets it to 0. The function is confusingly named, in that it has side effects. It also lacks documentation.

Status

Acknowledged.

B02: Unnecessary truncation of uint256

[Severity: Low | Difficulty: N/A | Category: Gas Cost]

In wOHM, whenever creating a new checkpoint (keeping track of an address' voting power at a given time) the block number is stored. wOHM passes the block number through a function called *safe32()*, which ensures the number is less than 2^{32}.

```
    struct Checkpoint {
        uint fromBlock;
        uint votes;
    }

    function _writeCheckpoint(address delegatee, uint nCheckpoints, uint
oldVotes, uint newVotes) internal {
          uint blockNumber = safe32(block.number, "wsOHM::_writeCheckpoint:
block number exceeds 32 bits");
          [...]
          checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber,
newVotes);
          [...]
    }
```

At the future planned block production speed of Ethereum (12 seconds per block, which is a somewhat faster than the current speed) it will take 1,600 years to exceed that number[9], making it a reasonable limitation.

However, there is no reason to perform this operation, and it increases gas cost. The reason that the code that wOHM is forked from (Compound's Comp.sol) uses *safe32()* is to pack struct tighter, using 32 bits per block number and 96 bits for the balances, giving 128 bits per struct in the checkpoint arrays, or 2 structs per EVM storage slot. wOHM does not use struct packing, and uses uin256 for both block number and balance.

Recommendation

Either remove the *safe32()* function and the above use of it, or use struct packing, making *fromBlock* a uint32 and *votes* a uint96 or uint224. The latter solutions require some refactoring, and that the *_balances* array is made into mapping from address to the same uint type.

[9] 2^{32} blocks * 15 s/block / 31,536,000 s/year ≈ 1,634 years.

Status

Addressed in commit [45f5fa531ea1a9532d93321a3f491784b39c314e](45f5fa531ea1a9532d93321a3f491784b39c314e).

B03: User delegation may block transfers

[Severity: Medium | Difficulty: Low | Category: Functional Correctness]

The current implementation of delegations means that on a *transfer, unwrap* or *wrap* transaction, the delegation is moved from the sender. If the sender has too little delegation, it will fail.

Scenario

User Alice has balance 10. She delegates to Bob, who has no previous delegation and no balance, giving him a voting power of 10, stored in his checkpoint 0. She now wraps some sOHM into wOHM, and is minted 5 wOHM. Alice is now delegated a power of 5 from the 0 address, stored in her checkpoint 0.

Alice now tries to transfer her 15 wOHM to Eve. The transfer function will try to move 15 voting power from Bob to Eve, but fail, and the transaction will revert without an error message. The same thing will happen if Alice tries to unwrap more than 5 of her wOHM.

Alice also cannot delegate to herself, or to the 0 address, because she can only delegate her full balance. The *_moveDelegates* function will always fail to move 15 votes from Bob. Her only option is to transfer her 5 tokens to some other address, or unwrap them, then undelegate from Bob.

Recommendation

See Recommendation under A04. This issue will not be present if the recommendations there are implemented.

Status

Like A04, addressed in commit 45f5fa531ea1a9532d93321a3f491784b39c314e.

B04: The term “bond” not used in its traditional sense

[Severity: N/A | Difficulty: N/A | Category: Documentation]

“Bonds” in traditional finance represent a loan. An entity wanting to borrow an asset sells a bond. They sell it for the currency they want. The price of the bond is fixed by the seller to the amount they want to raise. When the bond matures, the buyer of the bond will receive their payment back, plus some interest.[10]

In OlympusPro, “bonds” give no interest offered, the price is variable, the purchase currency and the currency paid back are different, and maturation happens gradually. The OlympusPro bonds therefore do not fulfill the definition of bonds, their closest similarity being that they have a maturation period instead of being a spot swap.

The available documentation makes liberal use of the “bond” term, assuming its meaning is understood. This may be a point of confusion in the documentation to any new user who is familiar with the traditional concept of bonds, and who is trying to grok what OlympusPro bonds are. They may make faulty assumptions about the behavior of the CustomBond contract (or rather its interface) based on their understanding of bonds, and may make mistakes or reject the investment opportunity based on this faulty understanding.

Recommendation

Offer an explanation in documentation clarifying that the term “bond” does not refer to the traditional finance term, but instead carries its own meaning, having no traditional retail finance counterpart.

Status

Acknowledged.

[10] https://www.investopedia.com/terms/b/bond.asp

B05: Anyone can deposit and redeem for anyone

[Severity: Low | Difficulty: Low | Category: Integrations]

Anyone can deposit for anyone, and anyone can initialize a redeem for anyone. This is important for integrations to keep track of. They cannot rely on being able to control the amount of deposited or redeemed bonds, or the amount of time they have left before their bond vests.

Scenario

An external contract uses OlympusPro bonding. It assumes that it will call *redeem()* by itself, read the return value and update its internal bookkeeping accordingly, keeping track of its number of redeemed tokens. It is now vulnerable to an attack where someone else redeems for the protocol, paying only gas cost, but possibly causing the tokens to get stuck in a contract that doesn't understand it has received them. If the contract relies on being able to call *redeem()* to get its outstanding tokens, it may suffer griefing, because it will not be able to proceed past the *redeem()* function.

Recommandations

Other protocols using OlympusPro as a lego need to understand the above, and not make assumptions about being able to control their own deposit and redeems.

Status

Acknowledged.

B06: Adjustments to the control variable can shoot past the target

[Severity: Low | Difficulty: N/A | Category: Functional Correctness]

That the control variable adjustment has not yet passed the target is only checked before the adjustment. This means the control variable can shoot past the target, with as much as 3% (the max per-block adjustment). If it's a one-time adjustment, -- , i.e., the $|currentCV - targetCV| \leq adjustmentRate$ -- then the target is ignored.

Status

Acknowledged.

B07: Control variable is adjusted after deposit

[Severity: Low | Difficulty: N/A | Category: Functional Correctness]

The control variable adjustment happens after each deposit is complete. This means that in situations that require quick adjustments, such as sudden price increases of the payout token, the intended adjustment is lagging.

Status

Acknowledged.

B08: Bond contracts can be reinitialized

[Severity: Medium | Difficulty: N/A | Category: Input Validation]

Bonds can be reinitialized (initialized more than once), if no deposits come in during a full vesting period. In this scenario, the current debt drops to 0, which allows reinitialization by the contract owner.

It is also possible for the owner to stop deposits by setting a low *maxPayout*. When reinitializing the owner has more freedom to set protocol parameters than they usually do.

- *vesting* can be set to any number. When changing it normally it is required to be more than 10,000 blocks. If vesting is set to 0, then users can take out bonds but can never redeem them.
- *maxPayout* can be set to any number. When changing it normally it is required to be less than 1 percent (of the total supply).

Status

Acknowledged.

B09: *trueBondPrice()* is an approximation that overshoots

[Severity: N/A | Difficulty: N/A | Category: Functional Correctness]

The real price is *bondPrice()*/(1 - fee), but the price stored as *truePricePaid* is *bondPrice()**(1 + fee). The “true bond price” is a little smaller than the actual price. The “true bond price” is used for slippage tolerance, so a user may end up setting their slippage slightly lower than necessary.

Recommendation

Change the calculation of *truBondPrice*, taking care to use a fixed point library or sufficient decimals to make the division precise enough for the intended purpose.

Status

Acknowledged.

B10: The *truePricePaid* is only for the last deposit

[Severity: N/A | Difficulty: N/A | Category: Functional Correctness]

The *truePricePaid* stored in a bond is the (approximation) of the actual price paid, stored in each bond and used to check price slippage when buying a bond, but not otherwise used in the protocol.

However, it is not trustworthy for informative purposes, because it is always updated to the price paid at the last time of deposit, and as such is not representative of actual true price that the user has paid for their entire bond, in case they top up their bond before fully redeeming it.

Recommendation

Change the stored *truePricePaid* to be the average price paid for an entire bond:

$$\frac{trueBondPrice() \cdot payout + bondInfo[\_depositor].truePricePaid \cdot bondInfo[\_depositor].payout)}{(payout + bondInfo[\_depoistor].payout)}$$

Status

Acknowledged.

B11: Tier ceilings array can be incorrectly constructed

[Severity: Low | Difficulty: N/A | Category: Input Validation]

There is no check that the tier ceilings array is correctly constructed. A partner may construct an incorrect array, where a later tier has a lower tier ceiling, causing them to more or less to Olympus than intended. A partner may have assumed that the ceilings were cumulative, for example, whereas they are in fact numbers for an absolute number of bonded tokens.

Recommendation

Either check at contract creation that each ceiling is strictly larger than the preceding one, or make the ceiling cumulative, so that they specify how many tokens should be additionally added before the current ceiling is surpassed.

Status

Acknowledged.

Security properties verified

CustomTreasury

Funds of principal tokens in the treasury are safe and under the control of partners. The only way to withdraw principal tokens from the CustomTreasury is by controlling the governance address and calling *withdraw*.[11]

Payout tokens are also safe in the treasury, as long as the partners only give the status of "bond" to those contracts deployed with the CustomBond code. They can only be withdrawn with the *withdraw* function by the governance address, and other addresses can only access them via the *deposit* function, which is only callable by contracts marked as "bond" in the treasury. Only the governance address can mark an address as a bond. In this sense, payout tokens can only be purchased by buying bonds, at which time they are sent to the bond contract. They can only be withdrawn via the *redeem* function, which will only transfer tokens if there is an open bond, which is only the case if it was bought through deposit at the price of that time. Here we assume that any price accepted by a bond contract is a fair price. Discussions of price security and market manipulation have already been covered above.

wOHM

Therorem: *getPriorVotes(user, blockNumber)* will always return the correct number of votes.

Necessary condition 1: the *checkpoints[account]* mapping is sorted, in the sense that for $0 \leq i < j < nCheckpoints[account]$ we have that $checkpoints[account][i].fromBlock < checkpoints[account][j].fromBlock$. This means both that *fromBlock* increases with higher indices, and that there are no duplicate values of *fromBlock* for any address. This holds, because *nCheckpoints[account]* starts as 0, and all subsequent modifications to *checkpoints* happen in the *_writeCheckpoint()* function, which preserves this property.

Necessary condition 2: the algorithm in *getPriorVotes* will always locate the checkpoint of the user which has the largest block number which is smaller than *blockNumber*, or return 0 if no such checkpoint exists (indicating the user had no balance at the time). Proof: See appendix, an inline proof is provided in the code.

[11] This assumes no vulnerability in the token contracts or in the address controlling the treasury.

Appendix 1: Correctness of *getPriorVotes*

A formal proof of the correctness is inlined as comments to the code:

```
/// Theorem: The algorithm returns as RESULT the votes checkpoint of account
with the largest fromBlock which is less than or equal to blockNumber.

/// Conditions: The entries checkpoints[account][i] are sorted in ascending
order on fromBlock for all 0 <= i < numCheckpoints[account], and contains at
most one checkpoint per fromBlock value except for 0.
/// It also contains no checkpoints with fromBlock larger than blockNumber.
/// Finally, for n >= numCheckpoins[account],
checkpoints[account][n].fromBlock = 0.

function getPriorVotes(address account, uint blockNumber) external view
returns (uint) {
    require(blockNumber < block.number, "wsOHM::getPriorVotes: not yet
determined");

    uint nCheckpoints = numCheckpoints[account];

    if (nCheckpoints == 0) {
        /// Correctness argument: the checkpoints[account] array is empty, so
no checkpoint exists, so RESULT should be 0. Alternatively, we can equate
this to there being a single checkpoint in checkpoints[account] with a 0 vote
value.
        return 0;
    }

    // First check most recent balance
    if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
        /// Correctness argument: checkpoints[nCheckpoints - 1] is the
largest number in the array, and is less than blockNumber, so RESULT is
correct.
        return checkpoints[account][nCheckpoints - 1].votes;
    }

    // Next check implicit zero balance
    if (checkpoints[account][0].fromBlock > blockNumber) {
         /// Correctness argument: checkpoints[0] is the smallest element, so
no element less than or equal to blockNumber exists, and RESULT should be 0.
        return 0;
    }

    /// Due to the above conditions, we now have:
    /// nCheckpoints >= 1
```

```
    /// checkpoints[account][0] <= blockNumber
    /// checkpoints[account][nCheckpoints - 1].fromBlock > blockNumber
    /// By assumption, 0 is the first index in the checkpoints[account]
“array” and nCheckpoints - 1 is the last index.

    uint lower = 0;
    uint upper = nCheckpoints - 1;

    /// Next we perform a binary search.
    /// Loop invariants:
    /// lower <= upper
    /// checkpoints[account][lower] <= block number
    /// checkpoints[account][upper] >= block number
    /// For n > upper, checkpoints[n] > blockNumber if it exists
    /// For n < lower, checkpoints[n] < blockNumber if it exists
    ///
    /// Each is trivially true at the outset.

    while (upper > lower) {
        uint center = upper - (upper - lower) / 2; // ceil, avoiding overflow
        /// Lemma: lower < center <= upper
        /// Proof: if center > upper, then upper - (upper - lower) // 2 >
upper, so (upper - lower) // 2 < 0, so lower > upper, contradicting the loop
condition.
        /// If lower >= center, then upper - (upper - lower) // 2 <= lower,
meaning
        /// upper - lower <= (upper - lower) // 2, meaning either upper <
lower
        /// (impossible by loop condition), or lower upper - lower = 0 ==>
upper =
        /// lower, likewise impossible.

        Checkpoint memory cp = checkpoints[account][center];
        if (cp.fromBlock == blockNumber) {
            /// Correctness argument: cp has the largest fromBlock in
checkpoints[account] which is less than or equal to blockNumber, because
cp.fromBlock == blockNumber and all checkpoints have unique fromBlock. So
RESULT is correct.
            return cp.votes;
        } else if (cp.fromBlock < blockNumber) {
            /// Lemma: loop invariants are preserved.
            /// Proof:
            /// After the next line, lower <= upper because before
            /// lower < center <= upper
            lower = center;
            /// Now, checkpoints[account][lower] <= block number
            /// and for n < lower, checkpoints[n] < blockNumber because the
array is sorted and checkpoints[center] < blockNumber.
```

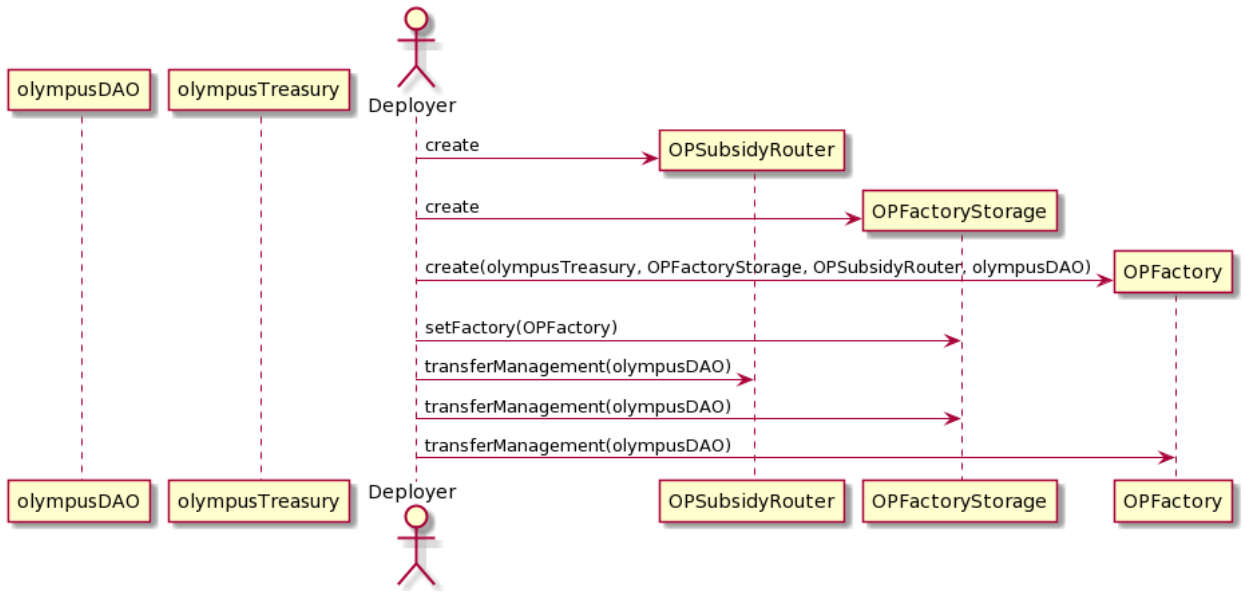
```
            /// The other invariants are unaffected.
        } else {
            /// cp > blockNumber
            /// Lemma: loop invariants are preserved.
            /// Proof:
            /// After the next line, lower <= upper
            /// because lower <= center - 1.
            /// Sub-proof:
            ///   Assume center - 1 < lower.
            ///   But center >= lower, so that means that center = lower.
            ///   That in turn means lower = upper - (upper - lower) // 2
            ///   which implies upper - lower = (upper - lower) // 2
            ///   which implies upper - lower = 0
            ///   which implies upper = lower.
            ///   That contradicts the loop condition,
            ///   so lower <= center - 1.
            upper = center - 1;
            /// Now, checkpoints[account][upper] >= block number
            /// For n > upper, checkpoints[n] > blockNumber because the array
is sorted and checkpoints[center] > blockNumber.
            /// The other invariants are unaffected.
        }
    }
    /// Lemma: the loop terminates.
    /// Proof:
    /// Every iteration, if the function does not return then
    /// either lower increases in value or upper decreases.
    /// This is given by the fact that at the outset
    /// lower < center <= upper and either lower is set to center or upper is
set to center - 1.

    /// Correctness argument:
    /// upper == lower.
    /// Proof: lower <= upper (by invariant), and !(lower < upper),
    /// so lower = upper.
    /// By loop invariants:
    /// checkpoints[account][lower] <= block number and
    /// for all lower < n < nCheckpoints - 1,
    /// checkpoints[account][n].fromBlock > blockNumber
    return checkpoints[account][lower].votes;
}
```

Appendix 2: Deployment Procedure and Analysis

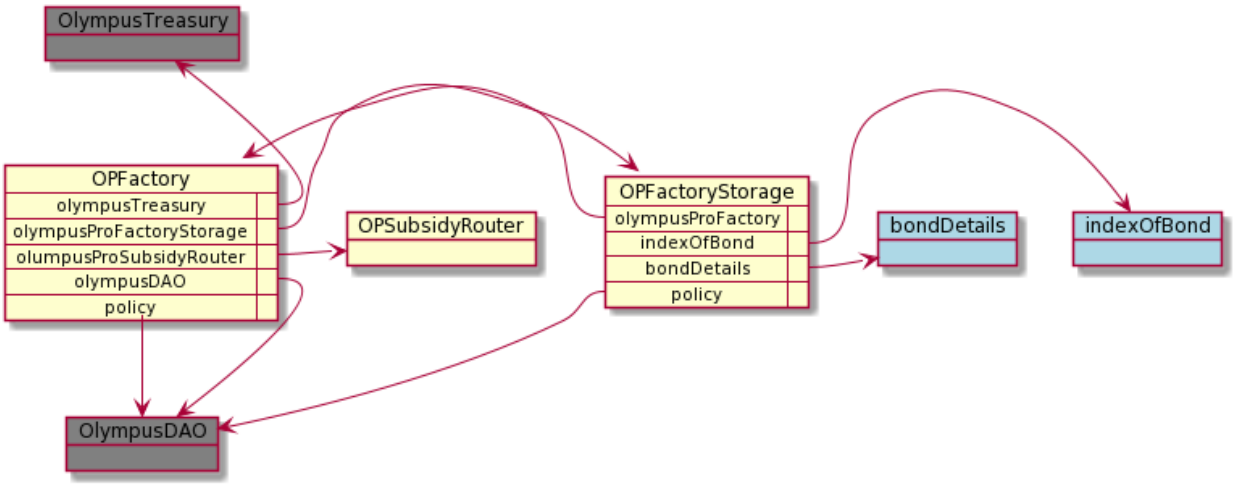
Deployment

The deployment process should look as follows, and it should be verified that all transactions go through.



Initial State

The initial state after deploying the factory, storage and deployment should be as follows. The grey objects are pre-existing addresses, the yellow objects are the OlympusPro core contracts, and the blue objects are (initially empty) maps.



Example State

Once deployed, the core contracts could have a state like the following one (indexes depending on deployment order).