Protocol audit report

The Thesis team asked us to review and audit the Tally protocol. We looked at the code and now publish our results.

Scope

We audited commit `556851a12b0d150a00782b1b2b504af6cca454c0` of the `tallycash/tally-contracts` repository. In scope are the smart contracts in the `contracts` directory. However the `test` directory was deemed out of scope.

Before overviewing the system and its privileged roles, and moving to a full list of issues found during the audit, some introductory remarks about the project's current status are in order.

Project status

We audited an early version of the Tally project that is a work-in-progress and not yet ready for production. In view of the project's maturity, this first security audit round should be taken as the initial step forward in the way to reach the highest levels of code quality and robustness demanded by systems intended to handle large sums of financial assets. We identified numerous opportunities for improvement in this code which are highlighted throughout the report. We note that there is no accompanying documentation, so there is little to go by for stakeholders to understand the facets of this system. This project will require not only specific patches in several code segments, but also efforts in terms of testing and the redesign of how components are architectured. While these issues could be considered symptoms of the inherent difficulty of building a sustainable complex financial system, by no means are they to be taken lightly. Further security review of the entire protocol are in order, which

along with our recommendations in this report, should help bring the project to a production-ready state.

System overview

The Tally protocol consists of various Defi services bundled into a single wallet, including yield farming and swapping against liquidity pools. The smart contracts defining this protocol are to be launched by users in tandem with a Defi enabled wallet for Ethereum, which will be the main interface to these services, but the wallet itself is not in scope.

`Tally` token

The `Tally` token and governance are forked from Compound's `COMP` token and corresponding GovernorBravo, with some significant modifications. One of the main additions to the `Tally` token is a mechanism that restricts accounts in their ability to transfer tokens to control distribution during the bootstrapping process. These account restrictions can then be removed permanently either individually or globally by a privileged role. The second major addition to the `Tally` token and its governance is a refactoring of their constructor logic to allow for a second initialization step to be more compatible with their deployment mechanism. The token will be constructed with an initial supply which can later be increased through minting by a privileged role. Like the `COMP` token, much of the utility of `Tally` is derived from it being used to delegate votes in the governance process.

Having forked from Compound, the Tally's `GovernorBravoDelegate` uses the same values of governance parameters as the Compound's `GovernorBravoDelegate`. The parameters include proposal threshold, voting period, voting delays and the number of votes needed to reach quorum. We expect these parameters to be altered depending upon how the Tally team envisions the distribution of Tally tokens. For example, if the distribution is slow, the current high value of `quorumVotes` could prevent the system from reaching a quorum.

Deployment mechanism

The effort in bootstrapping the Tally protocol through deployment of its smart contracts is to be distributed amongst users. The `TallyDeployer` contract defines a mechanism incentivizing user participation in this process, and ensuring proper initialization. The design of this mechanism leverages the deterministic addresses provided by the EIP-2470 singleton factory to properly map deployer incentives and to facilitate discoverability. Tally protocol contracts to be deployed in this manner are "whitelisted" by hardcoded assignments of contract rewards to corresponding target deployment addresses. Even the `TallyDeployer` itself will be deployed in this manner where the deployer will receive a corresponding reward in `Tally` token.

Hunting Grounds

The Hunting Grounds are incentivized pools that deploy their capital to other yield opportunities. The featured Hunting Ground is the Unipool Hunting Ground which deploys its underlying assets to another

Unipool to earn rewards. Users can stake and withdraw the featured token of the pool, and can harvest accumulated rewards in a reward token. These Hunting Grounds are to be deployed by the `TallyDeployer`, however as of this audit such Hunting Grounds are not yet "whitelisted" by the `TallyDeployer`.

Swapping

The `Swap` contract enables users to fulfill 0x quotes via its external `swapByQuote` convenience function. For each swap a `swapFee` is charged and the accumulated fees can only be swept by the `feeRecipient` which is set by governance. The `Swap` contract is to be deployed via a delegate pattern from governance, not by the `TallyDeployer`.

Treasury Vesting

The `TreasuryVester` contract is responsible to put an amount Tally tokens under a vesting schedule to a specific recipient. This can be deployed by any user with the right input parameters and must be funded with `vestingAmount` of Tally tokens after deployment.

Airdrop of Tally tokens

The `ExpiringMerkleDistributor` contract aims to manage the airdrop of Tally tokens through a Merkle distribution. It works by hashing airdrop participants together in a Merkle tree and to publish the merkle root of it in the smart contract. User can `claim` their airdropped tokens by providing Merkle proofs checked against the root. Additionally, this contract has an expiration time which enforces the users to claim their airdropped tokens before endTime is reached. Finally, the contract is also `Ownable` and the `owner` is the address that deploys it. After the expiration, any leftover tokens are transferred to the `owner` by the privileged `clawBack` function.

Roles and their security assumptions

The `Tally` token has a `minter` and an `unpauser` which are both set to the `msg.sender` of the `initialize` function. The `minter` can mint `Tally`, `removeAccountRestriction`'s, and `disableTransferRestriction` globally. The `unpauser` has all the privileges of the `minter` with the exception that it cannot mint `Tally`. Both of these roles can reassign their role to another account. It's the intent of the system design that the `TallyDeployer` will have these roles since it is to call `initialize`, however it is noted in the report that isn't necessarily the case.

The Hunting Grounds are `Ownable` where the owner has exclusive privileges to `pause`, set fees, and deallocate the underlying tokens, as well as toggle whether only EOA's or all accounts can stake. There is also a `rewardDistribution` role inherited from Synthetix's Unipool set by the pool owner, which is the only account able to `notifyRewardAmount`.

The Swapping pools are `EmergencyPausable` meaning they can be paused or unpaused only by the timelock or emergency governance.

The governance itself contains a role `emergencyGovernance` which can only be set by the timelock.

All external code and contract dependencies were assumed to work correctly. Additionally, during this audit, we assumed that the system administrators are available, honest, and not compromised.

The code base has been audited during the course of four weeks by three auditors and here we present our findings.

Critical severity

[C01] Admin is never set

The `TallyDeployer` contract is meant to be the one that users will need to interact with to deploy and initialize contracts by the protocol. This is because the system is meant to be deployed by users. To do so, the Tally team forked and modified other protocol contracts to adapt them to their intentions. Specifically the governance module, forked from Compound, has been adapted by removing some automations which are now manually triggered.

Specifically, the `GovernorBravoDelegate`, from which the `TallyGovernorBravoDelegate` contract extends, in the original Compound version, had a requirement for the `admin` to be the `msg.sender`, while Tally versions require `admin` to be unset at `initialization` time. `GovernorBravoDelegate`, and so `TallyGovernorBravoDelegate` works behind a proxy that is the actual storage layer, the `GovernorBravoDelegator`, which is a fork of Compound but with some modifications. The original Compound proxy had `admin` assignment and initialization in its constructor, while the forked version of `Tally` removes this automation, to let the `initialize` function to be called by a user later. For this, the `admin` assignment in the proxy's constructor has been removed, and placed instead inside the `initialize` function, but a bug has been introduced.

The `admin` is set to be the `timelock`, the state variable which isn't initialized until after the `admin` assignment. This way the `admin` is set to be the zero-address. It appears to be the intent of the developer that instead of `timelock`, the `admin` meant to be set to the `timelock_` passed input parameter instead.

This has catastrophic consequences in the system since:

- `admin` can't be changed anymore since, `_setPendingAdmin` can't be called.
- `_setProposalThreshold`, `_setVotingPeriod` and `_setVotingDelay` can't be called anymore.

Consider setting the `admin` parameter to the `timelock_` variable to resolve this issue.

[C02] Anyone can change governance implementation

As mentioned in [C01], because of a bug in `GovernorBravoDelegate` contract, the `admin` parameter ends up being the zero address. Apart from the already mentioned issue and its consequences, there is a related important issue that deserves its own description.

The `GovernorBravoDelegator` contract, being the proxy and storage layer for the `GovernorBravoDelegate` contract, has the `_setImplementation` function which can only be called either by the `admin` or by anyone when `admin == address(0)`.

This means that, at any moment, anyone is able to call the `_setImplementation` function and change the implementation address of the `GovernorBravoDelegate` contract, potentially to a contract that executes malicious code.

The fact that the `_setImplementation` function is having the possibility to be called when `admin == address(0)` is because of initialization purposes, and the fix to the already mentioned issue should solve this issue too.

Alternatively, consider redesigning the `_setImplementation` function to not use the null admin condition to correctly process a call.

[C03] Contract deployments can be "jacked"

The `TallyDeployer` contract is an intermediary for deployment and initialization of known Tally contracts where the deployer is rewarded in Tally token. The first instance of a contract being deployed in this manner is the `Tally` token itself, whose deployment is triggered within the `TallyDeployer` constructor. The `TallyDeployer` is expected to be deployed by the EIP-2470 singleton factory, and the `tx.origin` will receive the deployer reward. The other instances are the `TIMELOCK`, `DELEGATOR`, and `TallyGovernorBravoDelegate` whose deployments are triggered within the `TallyDeployer` `deploy` function. In this latter case the `msg.sender` will receive the deployer reward.

This deployment mechanism leverages the deterministic addresses provided by the singleton factory to map target contract deployment addresses to contract rewards, and aid in discoverability regardless of who the deployer is. The singleton factory's underlying `create2` call is a deterministic function of the caller, salt, and initialization bytecode which includes set constructor parameters. A subtlety that must be noted is that the caller considered by the `create2` function is the `msg.sender` of the `create2` function. In the case of the singleton factory, the `msg.sender` to the `create2` function is the singleton factory itself. This means that for a given salt and initialization bytecode, the singleton factory will give the same address regardless of which contract or EOA calls it. A final thing to note is that `create2` will fail if the target address already contains deployed contract code, so a given deployment can only happen once.

The singleton factory can be called directly by any account with the salt and initialization bytecode corresponding to any of the target deployments of the `TallyDeployer` contract. This way a target address that hasn't yet been deployed can be "jacked" by a deployment circumventing the `TallyDeployer` contract.

There are many issues stemming from this. The least severe of which is that a jacked deployment will never pay its corresponding deployer rewards. But more severe problems come from the fact that initialization of these contracts is now outside the `TallyDeployer`'s control:

- The `msg.sender` calling `initialize` on the `Tally` token is set as its `minter`, and `pauser`, and has the `totalSupply` minted to it. The intended system design is so that the `TallyDeployer` will have these roles and this balance since it calls `initialize` within the same transaction where it deploys the `Tally` token. But, a malicious actor can frontrun construction of the `TallyDeployer` by deploying the `Tally` token through the singleton factory so that within the same block, anyone's `initialize` transaction could be mined and they would seize full control of the token. Also note that since the `Tally` token address is jacked, the constructor of the `TallyDeployer` will now always revert since it checks that the `TALLY` address does not contain deployed bytecode. So this attack locks out the `TallyDeployer` from the Tally protocol.

- The `TallyDeployer` has special logic initializing the `DELEGATOR` along with its deployment. This initialization sets the timelock, tally token, and other important parameters. Again, a malicious actor can deploy the `DELEGATOR` through the singleton factory, exposing the `initialize` function to be called by anyone. This way, the timelock, token, and other critical parameters can be set to malicious values. This can have many devastating effects on the system. For instance, if an attacker picks the `timelock` to be under its control it can arbitrarily manipulate the queueing and executing of proposals in governance.

Consider hardcoding parameters that are otherwise set by initialization functions of contracts deployed in this manner. Moreover, being able to circumvent the `TallyDeployer` functionalities, is an intrinsic outcome of the chosen design, for this, we recommend reviewing the pattern chosen and decide to either ease the requirements on the functionality or to strengthen the design to reflect the real intents of the protocol.

[C04] Deployment scheme blocks core `Tally` token functionality

The ERC20 compliant `Tally` token is the governance token in the Tally protocol. The `Tally` token is a fork of Compound's `COMP` token, and both tokens are purpose-built for their respective governance protocols. In fact, the Tally protocol's governance is also forked from Compound's GovernorBravo, so the design choice in forking the `COMP` token comes naturally.

The `Tally` token does have additional functionality not present in the `COMP` token.

The main addition is a mechanism to restrict accounts' ability to exchange `Tally`. This is implemented simply as a check of the spender in the global `unrestrictedAccounts` mapping within the `transfer` and `transferFrom` functions. All accounts are initialized as being restricted in this way, except for the `minter`, whom along with the `unpauser`, can call the `removeAccountRestriction` function on behalf of an account. There is also a `disableTransferRestriction` function callable only by the `unpauser` that when called, will permanently disable this mechanism, allowing all accounts to transact tokens freely.

Another addition is that the `Tally` token has an `initialize` function that must be called in the same block that the contract is constructed. Here, the `minter` and `unpauser` roles are set to be the `msg.sender`, the `minter` is set to be an unrestricted account, and the `minter` is given the balance of the `totalSupply`.

The `Tally` token is to be deployed by the `TallyDeployer` contract by way of the EIP-2470 singleton factory. This deployment will in fact occur within the constructor of the `TallyDeployer` which will immediately call the `initialize` function on the newly deployed `Tally` contract. This means that the `TallyDeployer` will be assigned the `minter` and `unpauser` roles of the `Tally` token since it is the `msg.sender` of `Tally`'s `initialize` function. Lastly a deployment reward is transferred to the `tx.origin` of the deployment of the `TallyDeployer`. We note here that there are currently two holders of `Tally` token: the `TallyDeployer` and the recipient of the deployment reward.

Aside from the aforementioned initialization and transfer, the `TallyDeployer` does not expose any methods directly or indirectly of the `Tally` token contract.

This means that the `TallyDeployer` cannot transfer `Tally` token to any other accounts since the `transfer` and `transferFrom` methods are not available. For any account besides the `TallyDeployer` holding `Tally`, which we know is solely the reward recipient above, their ability to transfer `Tally` is restricted and cannot be resumed since both the `removeAccountRestriction` and `disableTransferRestriction` functions are inaccessible to the `TallyDeployer`.

From here we see `Tally` cannot be minted, the `unpauser` and `minter` cannot be set, `Tally` can't be transferred since all these functions are inaccessible to the `TallyDeployer` and the reward recipient is restricted. A final interesting note is that only the reward recipient can delegate votes in the governance process since the only other holder is the `TallyDeployer` contract, whom we know now, is incapable of interacting with the `Tally` token.

Within their test suite, the Tally team tests some functionality of both the `Tally` token and `TallyDeployer` as they work together, but these tests are not comprehensive enough and do not faithfully simulate the bootstrapping of Tally's economy.

Consider implementing a wrapper to the `TallyDeployer` that properly exposes the functions of the `Tally` contract. Furthermore, consider fortifying this implementation with a comprehensive test suite taking into consideration the subtleties of the context of the calls, and a more thorough simulation of bootstrapping this economy.

[C05] Withdrawal from hunting grounds does not account for reward update

The `HuntingGround` contract inherits `StakedTokenRewardPool` contract. When a user stakes through a hunting ground, the `stake` function in `StakedTokenRewardPool` contract is invoked. This parent `stake` function calls the `updateReward` modifier which updates the values of reward per token and the reward accumulated by the user.

Similarly, if a user interacts with the `withdraw` function of the `StakedTokenRewardPool`, the `updateReward` modifier is called and respective values are updated. However, the same does not happen if the user withdraws from the hunting ground.

The `withdraw` function in the `HuntingGround` contract overrides the parent `withdraw` and does not invoke it. Additionally, this function does not account for the update of rewards. Therefore, every time a user interacts with this `withdraw` function, they miss out on some reward, since rewards accrued by their account are not updated.

Moreover, this also impacts the reward for all the users. Since the `updateReward` is not called, each call to this `withdraw` function would miss the increment to `rewardPerTokenStored`, which affects all the users interacting with the hunting ground.

To sum it up, the more users withdraw from the hunting ground, the lesser rewards are accrued to all the users since reward per token is not incremented the way it should.

This problem can be solved if the user calls `getReward` function prior to withdrawing. However, since there is no prerequisite to calling `withdraw` at the moment, any user can interact with it and mess up with the rewards.

Consider changing the design of `withdraw` function to either adding prerequisite to calling the function or accounting for the proper update of rewards.

[C06] Withdrawal fee is locked inside `HuntingGround`

The users, who interact with the hunting grounds, need to pay a cut of their rewards as the `performanceFee` and a portion of their `wrappedToken` as the `withdrawalFee`. The `HuntingGround` contract defines a `groundFees` mapping which keeps a track of these fees paid by each user account.

While the `groundFees` is correctly tracking the collection of `performanceFee`, it does not track the `withdrawalFee` paid by an account.

The `sweepFees` function is defined in the `HuntingGround` contract which, when called, transfers an amount of accrued fees to the owner of the `HuntingGround` contract. Within this function, before the fee is transferred, the input amount is deducted from the `groundFees`. Since there is no mapping of `wrappedToken` tokens in `groundFees`, this subtraction will underflow thereby reverting the function call and locking the `withdrawalFee` in the HuntingGround contract.

Consider implementing the mapping of `withdrawalFee` within the `withdraw` function.

High severity

[H01] Hunting ground fees can be stolen

The `withdraw` function of the `HuntingGround` contract allows a user to make a request to withdraw "`amountUnderlying`" of `wrappedToken` from the hunting ground. This `withdraw` function is available in the `UnipoolHuntingGround` contract as well since it inherits the `HuntingGround` contract. The `HuntingGround` accumulates fees in the `wrappedToken` and the `tokensEarned` by taking a cut of transfers within the `withdraw` and `getReward` functions.

The routine of the `withdraw` function first has a `preWithdraw` step, then checks the `HuntingGround` has a balance in excess of the requested `amount`, and finally transfers the `wrappedToken` to the `msg.sender` while skimming its fee.

The `preWithdraw` makes an `attemptToDeallocateUnderlying` which, as implemented by the `UnipoolHuntingGround`, calls the `withdraw` method on its own `IStakingTokenRewardsPool` `farm` within a `try` `catch` block. This `attemptToDeallocateUnderlying` is designed to endow the `HuntingGround` with the necessary liquidity in `wrappedToken` to make the final transfer to the `msg.sender` of the `withdraw` routine.

The problem is as follows: If the calling of the `withdraw` method of the `farm` fails within the `try` `catch` block, the `attemptToDeallocateUnderlying` will return `false`. This boolean is also the return value of the `preWithdraw` function. But within the `withdraw` function of the `HuntingGround`, the return value of `preWithdraw` is never checked. This way, the transfer to the `msg.sender` of the `amount` less `fees` will be deducted from the `HuntingGround`'s balance which, in this case, is not supplemented with a withdraw from the farm but only consists of the accumulation of its own fees.

The farms to be used in these hunting grounds are expected to be outside this codebase, but are not yet known to be established. So analysis of their likelihood to fail within the `try` `catch` block of the `attemptToDeallocateUnderlying` is not certain. But it could be the case that a `farm` is chosen where a user can engineer circumstances where `withdraw` deducts from the accumulated fees without being supplemented from the `farm`.

Consider programming defensively by checking and reacting to the return value of the `preWithdraw` function within the `withdraw` routine.

[H02] Hunting-grounds fees can be stuck in owner

The `Ownable` `HuntingGround` contract accumulates fees by taking a cut of each `earnedAmount` when a staker collects their rewards by calling the `getReward` function. These fees are accounted for by incrementing the `groundFees` mapping for a given `tokensEarned` within the `getReward` function and represents the `HuntingGround` balance in these tokens in excess of what is transferred to the staker. The `HuntingGround` contract exposes an external `sweepFees` function, callable by any account, that will transfer the `groundFees` for a given `token` to the `owner` of the `HuntingGround` contract.

The `Hunting Grounds` of the Tally protocol are intended to be deployed by the `TallyDeployer`'s `deploy` mechanism. The `TallyDeployer`'s `deploy` mechanism deploys whitelisted contracts using EIP-2470 singleton factory. In using the singleton factory to deploy contracts, the `msg.sender` considered within the context of the deployed contract's constructor will be the caller of the `create2` function, which in this case is the singleton factory. Recall the `HuntingGround` is `Ownable` and does not at any point `transferOwnership` to any other account. This means that the `msg.sender` constructing the `HuntingGround`, the singleton factory, will be the `owner` and thus the recipient of the `HuntingGround`'s swept fees.

Consider adding to the `HuntingGround`'s constructor logic a `transferOwnership` to an administrative account within the protocol which itself has capabilities to withdraw or transfer funds it receives.

[H03] Tokens with uncommon decimals lead to incorrect rewards

The `HuntingGround` contract is distributing to the stakers all accrued rewards, from all the tokens in which they participated. This is done in the `getReward` function.

To do so, the function first calculates the rewards earned in `rewardTokens`, then it uses them to calculate the `earnedAmount` of each token and, after subtracting some fees, it sends the leftover of each token to the `msg.sender`.

The problem is that the contract, when calculating `earnedAmount`, is implicitly assuming that each token in the `tokensEarned` array are all scaled to 18 decimals since `rewardToken` is used in the first place.

The fact that the `rewardToken` has 18 decimals is itself an assumption, since it is not checked to be so when `rewardToken` is initialized, but it's assumed to be so when performing internal calculations.

If some of the tokens in the `tokensEarned` array is having a different number of decimals, or if the `rewardToken` itself doesn't have 18 decimals, rewards calculation can give unexpected results, making accounting wrong and lastly giving more or less rewards to users depending on the exact number of decimals.

Consider explicitly informing users, through the docstrings or documentation, about the process of setting farming tokens, taking into considerations the number of decimals but also uncommon token behaviours, like fees deducting tokens like USDT that may lead to incorrect and unexpected amounts being transferred.

Moreover, given the tight dependency between `rewardToken` and each one of the `tokensEarned`, consider explicitly calling the `decimal` public function of a standard `ERC20` token and require to the returned value to be 18.

Medium severity

[M01] Event issues

The following functions of the `Tally` contract do not emit relevant events after executing sensitive actions:

- The `disableTransferRestriction` function which disables all transfer restrictions on the token, and cannot be re-enabled.
- The `removeAccountRestriction` function removing transfer restrictions on an account.
- The `setUnpauser` function which updates the `pauser` role to a new address.

Also, many events defined in the contracts have no indexed parameters:

- The `Claimed` event of the `IMerkleDistributor` interface.
- The events defined in the `HuntingGround` contract.
- The events defined in the `Swap` contract.
- Many of the events defined in the `GovernorBravoInterfaces.sol` file.

Consider emitting events after sensitive changes take place, to facilitate tracking and notify off-chain clients following the contracts' activity. These events should be defined with indexed event parameters to avoid hindering the task of off-chain services searching and filtering for specific events.

[M02] Lack of input validation

There are many examples in the code base of lack of input validation. Some examples are:

- On line 174 of `Tally` contract, `minter_` is not checked to not be the zero address. The `mint` function can forever be locked in the case that the `minter` is set to an inaccessible address.
- On line 208 of the `Tally` contract, the `rawAmount` is not checked to be different from zero.
- On line 54 of the `TallyGovernorBravoDelegate` contract is not checking whether the `newEmergencyGovernance_` is not the zero address.
- On line 189 of `Tally` contract, the `unpauser_` parameter is not checked to not be the zero address. Since there is also a check that `msg.sender == unpauser` the `unpauser` role can forever be lost to the zero address. The `disableTransferRestriction` function explicitly sets the `unpauser` to the zero-address, so it is best that such a drastic setting is only done in this single `disableTransferRestriction` function and not accidentally in the `setUnpauser` function.

- On line 176 of the `HuntingGround` contract, the `amount` is not checked to be less than `groundFees[token]`, this would revert the subtraction in line 177 with no informative messages.

- The constructor of the `MerkleDistributor` contract is not checking whether the `token_` and the `merkleRoot_` parameters are non trivial.

- The constructors of the `Unipool.sol` contracts are not checking whether their input parameters are non trivial.

- The constructor of the `Swap` contract does not check that `swapFee` is within any bounds, while the `setSwapFee` function does. For consistency the constructor should check `swapFee` against the same bounds.

- The constructor of the `ExpiringMerkleDistributor` contract is not checking that `startTime` is a non trivial value or not a value in the past. If a value in the past is accepted, consider properly describing it in the docstrings.

Even though this issue does not pose a security risk, the lack of validation on user-controlled parameters may result in erroneous transactions considering that some clients may default to sending null parameters if none are specified.

[M03] Pending Admin can be set to `address(0)`

The system is meant to never lose the `admin` role in the governance contracts, so safety measures have been put in place preventing it from being assigned to `address(0)`. This is clearly the intention behind the docstrings in lines 553-554 of the `GovernorBravoDelegate` contract.

The reason is that if the `admin` would ever be set to `address(0)`, anyone would be able to call the `_setImplementation` function of the `GovernorBravoDelegator` contract, the proxy and storage layer behind the governor contract, because it would pass the check in line 22.

But, as implemented, the `actual` admin can call the `_setPendingAdmin` function of the `GovernorBravoDelegate` contract and set the `pendingAdmin` to the zero address.

This doesn't pose a security issue, since the `_acceptAdmin` function requires that the `msg.sender` is the `pendingAdmin` and is not the zero address. Even without that safeguard, it is highly improbable that any entity owns the private key of the zero address that would call this `_acceptAdmin` function.

The issue is that the checks for `msg.sender != address(0)` in lines 556 and 577 are meaningless, since the zero address effectively can't call this contract. What the checks should state is that `newPendingAdmin != address(0)` for line 556 and the second check in line 577 should be removed.

Even if this doesn't pose a security issue on its own, it's clearly a bug and a mismatch with the intended behaviour. Consider fixing both require statements according to the intended functionality.

[M04] Staking is possible before the call to `notifyRewardAmount`

The `StakedTokenRewardPool` contract defines the `lastUpdateTime`, `periodFinish` and `rewardRate` parameters that are initialized in the `notifyRewardAmount` function and are used to calculate a user's generated rewards.

If these parameters are not set, users will not generate any reward on their stakes, since any call to the `rewardPerToken` function will multiply the stored value of reward per token by the null `rewardRate` parameter, thereby making the accrued reward zero.

In an edge case scenario, where the users can call the `stake` function independently before any call is made to the `notifyRewardAmount` function, the staking will not accrue rewards.

Consider restricting the `stake` function to be called only if the `rewardRate` has been initialized, making sure that this change doesn't interfere with the expected design and behaviour of the contract.

Low severity

[L01] Array length can overflow loop's index parameter

The `for` loop within the `getReward` function of the `HuntingGround` contract has an index of type `uint8` and iterates over the `tokensEarned` array.

In the case that the length of the `tokensEarned` array is greater than `type(uint8).max`, the index will overflow. Although Solidity 0.8.0 will catch this overflow, the loop will eventually revert without any informative message.

Consider checking that the length of the array is not greater than `type(uint8).max` or changing the type of the index parameter to iterate over bigger array lengths. Alternatively, consider manually catching the overflow and revert with an informative explicit message.

[L02] Governance parameters are unchanged from Compound governance

The parameters hardcoded in `GovernorBravoDelegate` contract are identical to the Compound governance contract. While the value of most of these parameters depend on the type of distribution that Tally team envisions for the governance tokens, the name of this contract is initialised to `Compound Governor Bravo`.

Since this contract has been modified, to avoid any confusion, consider changing the name to reflect that it belongs to the Tally project and is not the same as Compound's.

[L03] Empty `try` block

The `attemptToDeallocateUnderlying` function in the `UnipoolHuntingGround` contract contains an empty `try` block and the logic to follow from the successful `try` is written after the entire `try/catch` block.

Programmatically, this implementation is equivalent to that of convention, but could affect readability or maintainability. Consider implementing the success of `try` statement inside the `try` block.

[L04] `HuntingGround` contract can't be unpaused

The OpenZeppelin's `Pausable` contract provides the `_pause` and `_unpause` internal functions, along with some modifiers that prevents other functions from being called when the system is either paused or unpaused. Since `_pause` and `_unpause` are internal functions, the child contract that inherits from them must implement wrapper functions to call them.

The `HuntingGround` contract inherits from the OpenZeppelinâ€™s `Pausable` contract and implements the `pause` function that prevents users from calling the `stake` function of a paused system. However, there is no implementation for the `_unpause` functionality.

Since this is a design choice of Tally team, consider properly documenting this immutable behaviour to keep the stakeholders informed.

[L05] Incorrect error messages

Some error messages in require statements are technically incorrect:

- L64 of `Swap.sol`
- L125 of `GovernorBravoDelegate.sol`

The wording of these error messages do not acknowledge the boundary of the interval being checked and are thus technically incorrect.

Error messages are intended to notify users about failing conditions, and should provide enough information so that the appropriate corrections needed to interact with the system can be applied. Uninformative or incorrect error messages greatly damage the overall user experience, thus lowering the system's quality. Therefore, consider not only fixing the specific issues mentioned, but also reviewing the entire codebase to make sure every error message is informative and user-friendly enough. Furthermore, for consistency, consider reusing error messages when extremely similar conditions are checked.

[L06] `initialProposalId` is never set

The `_initiate` function of the `GovernorBravoDelegateStorageV1` contract from which the Tally protocol forked their `GovernorBravoDelegate` contract was the only function to initialize the `initialProposalId` to a value.

Since this function has been removed, the `initialProposalId` parameter, together with the second check of the `require` statement in line 335, are useless now.

The purpose of this parameter was to enforce continuous values over proposal IDs across governance upgrades but the initial `Tally` governance will not need this parameter. However, if an upgrade is performed, this parameter, together with the appropriate value checks where needed, can be included in the upgrade itself.

To increase readability and have a cleaner code base, consider removing the `initialProposalId` and the mentioned check in the `require` statement.

[L07] Constant not declared explicitly

There is an occurrence of literal value in `TallyDeployer` contract that is not declared explicitly. Literal values in the code base make the code harder to read, understand and maintain, thus hindering the experience of developers, auditors and external contributors alike.

Developers should define a constant variable for every magic value used (including booleans), giving it a clear and self-explanatory name. Additionally, for complex values, inline comments explaining how they were calculated or why they were chosen are highly recommended. Following Solidity's style guide, constants should be named in `UPPER_CASE_WITH_UNDERSCORES` format, and specific public getters should be defined to read each one of them.

[L08] Missing docstrings

Some of the contracts and functions in the code base lack documentation. Additionally, the docstrings in some of the contracts do not follow the NatSpec, for example, the `EmergencyGovernable` contract uses `///` for multiple line comments instead of `/** ... */`. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

[L09] Multiple Solidity versions in use

Throughout the code base there are different versions of Solidity being used. For example, the `TallyDeployer` contract allows compiling with any version greater than 0.8.0 whereas the `SafeMath` library allows compiling with versions greater than 0.5.16.

Additionally, Timelock and TreasuryVester contracts are also using an older solidity version (^0.5.16). The Timelock contract calls a function that is deprecated in the newer solidity versions.

To avoid unexpected behaviors, all contracts in the code base should allow being compiled with the same Solidity version.

[L10] Overloaded functionalities

There are some parameters which are used for multiple purposes, overloading their initial one. Parameters should have clear names that reflect their functionality and they should have only one purpose inside the codebase.

Not following these guidelines, makes the codebase harder to read and understand, but it is also prone to errors and for this we don't recommend adopting it.

Specifically, in `Tally` contract:

- the `unpauser` global variable is used to whitelist users for transfers, but when unset, it is also used to signal that transfer restrictions are disabled.
- the `balances[address(0x0)]` variable is used to force construction and initialization to happen in the same block, but it is also a mapping used to track balances.

Consider defining one single variable for each of the mentioned functionalities, without overloading the existing one.

[L11] `Tally` doesn't inherit the appropriate interface

Within the `deploy` function of the `TallyDeployer` contract the `TALLY_TOKEN` is used as second parameter to the `DELEGATOR`'s initialize function. This initialize function sets its global variable, `comp`, to be the value of this second parameter cast as type `CompInterface`.

The current state of the code has the `Tally` token correctly implementing the `CompInterface` in that its `getPriorVotes` function conforms to the `CompInterface`'s corresponding function signature. But any updates to the codebase around the `Tally` token's `getPriorVotes` function could introduce bugs where its function signature differs from that of the `CompInterface`. This would affect its use in the governance process such as submitting proposals and casting votes.

Consider inheriting the appropriate contracts to reinforce implementation of interfaces, so that such bugs can be caught at compile time.

[L12] Unclear behaviour of `ExpiringMerkleDistributor`

The `ExpiringMerkleDistributor` contract allows anyone to call the `claim` function as long as the `block.timestamp < endTime` and allows the `owner` to call the `clawBack` function whenever `block.timestamp > endTime`.

It is not clear whether the contract should allow users to call the `claim` function or allow the `owner` to call the `clawBack` function when `block.timestamp == endTime`.

In order to improve correctness and consistency but also to increase completeness of the codebase, consider specifying which is the intended behaviour in this case and change the codebase to better represent it.

[L13] Unnecessary use of `SafeMath`

The majority of the smart contracts present in the code base are using a Solidity version which includes built-in functionalities to protect from overflows.

For this reason, the battle-tested OpenZeppelin's `SafeMath` library is not needed anymore and no special wrapping around `require` statements must be placed when performing arithmetic operations.

However, the code base is still using safe math operations to do calculations:

- The `Swap.sol` and `Unipool.sol` files are using `SafeMath` for `uint256` but contracts are compiled using Solidity `^0.8.0`.
- The `GovernorBravoDelegate` contract is using specific functions to handle arithmetic operations, but again the contract is compiled using Solidity `^0.8.0`.
- There is a `SafeMath` contract under the `external` directory that is imported in `Timelock` and `TreasuryVester` contracts. This is because these contracts are compiled using an old Solidity version. If those contracts were refactored to use latest Solidity version, the entire `SafeMath.sol` file could be removed.

There is an important difference between the Solidity built-in functionalities and the `SafeMath` library in that the built-in Solidity funcionalities use an invalid opcode to revert, consuming all the remaining gas of the transaction, while `SafeMath` uses `revert` opcode, leaving remaining gas untouched. However to override the built-in Solidity functionality, developers must make use of the `unchecked` keyword and then use `SafeMath` functions as intended.

To avoid using unnecessary functions, consider removing the use of `SafeMath` where not needed. If gas efficiency is important for the protocol, at the cost of added complexity, consider overriding Solidity built-in functionality to avoid consuming all gas in a possible arithmetic overflow.

[L14] Zero amount can be used in `withdraw` function

The `HuntingGround` contract inherits from the `StakedTokenRewardPool` contract and overrides its `withdraw` function.

The business logic of the overridden `withdraw` function in `HuntingGround` is different by design, hence it does not make a call to the parent function. However, this implementation of `withdraw` lacks certain checks, such as the one on the value of the passed `amount` input parameter, which is restricted in the parent function. Calling the `withdraw` function by passing 0 amount will result in triggering of events and wastage of gas. Within this function, consider checking that the input amount is greater than zero.

Notes & Additional Information

[N01] Erroneous docstrings and comments

Several docstrings and inline comments throughout the code base were found to be erroneous and should be fixed. In particular:

- On line 29 of `GatedStakedTokenRewardPool` contract, the comment `if the pool is gated` should be `if the pool is not gated` to reflect the exact behaviour of the function to which it refers.
- On line 48 and 51 of `Tally` contract, informal comments are used instead of NatSpec docstrings.
- On line 109 of `HuntingGround` contract, `amountUnderlying` should be `amount`.
- On line 115 of `TallyDeployer` contract, `+ Tally token deployment` should be removed since it is actually deployed in line 99.

[N02] Erroneous test

The test case in line 408 of the `SwapTests.ts` file is failing.

As the test suite was left outside the audit's scope, please consider thoroughly reviewing the test suite to make sure all tests run successfully after following the instructions in the README file.

[N03] Gas optimization

A possible gas cost improvement was found in the `castVoteBySig` function of the forked `GovernorBravoDelegate` contract where the `domainSeparator` is a function of constant values and thus can be computed once and set as a global variable in the constructor.

[N04] Inconsistent format in error messages

Error messages throughout the code base were found to be following different formats. In particular, some messages are formatted "Contract name::function name: error message", whereas others are not. Moreover, the error messages in `GovernorBravoDelegate` contract states the contract name as "*GovernorBravo*" instead of "*GovernorBravoDelegate*".

So as to favor readability and ease debugging, consider always following a consistent format in error messages.

[N05] Integer operations are not explicitly casted

The current lack of explicit casting when handling unsigned integer variables in the `HuntingGround` contract hinders code's readability, making it more error-prone and hard to maintain.

An example of this issue can be found at calculation of withdrawal fees, where multiplication of `uint256` and `uint128` is divided by `uint128`.

Consider explicitly casting all integer values to their expected type when sending them as parameters of functions and events. It is advisable to review the entire codebase and apply this recommendation to all segments of code where the issue is found.

[N06] Missing license

The `Unipool.sol` file contains a lot of contracts, and unlike other contract files, is missing an SPDX license identifier. Instead it defines a `MIT License` docstring.

To silence compiler warnings and increase consistency across the codebase consider adding a license identifier. While doing it consider referring to spdx.dev guidelines.

[N07] OpenZeppelin Contract's dependency is not pinned

To prevent unexpected behaviors in case breaking changes are released in future updates of the OpenZeppelin Contracts' library, consider pinning the version of this dependency in the package.json file.

[N08] Solidity compiler version is not pinned

Throughout the code base, consider pinning the version of the Solidity compiler to its latest stable version. This should help prevent introducing unexpected bugs due to incompatible future releases. To choose a specific version, developers should consider both the compiler's features needed by the project and the list of known bugs associated with each Solidity compiler version.

[N09] Readability issues

Unnecessarily verbose parameter names

The `swapByQuote` and `fillZrxQuote` functions of the `Swap` contract has unnecessarily verbose parameter names, making the code difficult to read.

It is understood that these parameters are to be aligned with the returned quote from the 0x quote API. Given this context is understood, the "zrx" prefix for each parameter adds clutter to the code.

Consider dropping the "zrx" prefix from the names of the parameters to these functions.

Duplicated code

The `swapByQuote` function of the `Swap` contract has a 14 line code block for each of two cases.

These code blocks are identical with the exception of the `boughtERC20Amount`/`boughtETHAmount` variables, and an ERC20 transfer.

This duplicate code makes understanding the code more difficult for stakeholders.

Consider assigning an intermediate "`boughtAmount`" variable depending on the two cases where the ERC20 transfer is made in the appropriate case so that the bulk of the logic doesn't need to be repeated.

[N10] Renaming suggestions

Good naming is one of the keys for readable code, and to make the intention of the code clear for future changes. There are some names in the code that make it confusing, hard to understand, or could otherwise be more precise.

Consider the following suggestions:

- `onlyTimelockOrEmergencyGovernance` to `onlyEmergencyGovernanceOrTimelock`
- `unrestrictedAccount` to `account`
- `rewardDistribution` to `rewardDistributor`

[N11] Style suggestions

In the code base, there are some lines that may benefit from a change in the style:

- There is an inconsistency in naming internal functions, where some function names start with `_` and others don't.

- Interfaces may benefit from being consolidated instead of being spread throughout the codebase.

- Functions that change Governance parameters like the `_setVotingDelay`, `_setVotingPeriod`, `_setProposalThreshold` and `_setPendingAdmin`, are requiring the `msg.sender` to be the `admin` while the `TallyGovernorBravoDelegate` contract is defining an `onlyTimelock` modifier to restrict access to the `setEmergencyGovernance` contract.

Consider applying a consistent style and to review the code base trying to optimize readability through style changes. This should improve readers understanding of the contracts.

[N12] Incomplete test suite

The unit tests provided in the code base are addressing the main pieces and functionalities of the system but they are not complete.

For example, tests for the `TreasuryVester` contract are not present and tests for governance don't include unit tests for proposal executions.

Consider reviewing the test suite and include tests for all the contracts present in the repository, including forked code from other protocols, making sure to establish at least a 95% coverage. Moreover, Continuous Integration systems are intended to run all unit tests of the project before merging any changes. This prevents introducing bugs into existing code, and helps keep the repository in a consistent tested state at all times.

Tally has no Continuous Integration setup, making it risky to introduce changes. Consider setting up a Continuous Integration system like CircleCI to run the unit tests on every pull request. Make sure that all tests are passing before merging any pull request.

[N13] Todo in code

In the `Swap` contract, there is a "TODO" comment that should be tracked in the project's issues backlog.

During development, having well described "TODO" comments will make the process of tracking and solving them easier. Without that information, these comments might tend to rot and important information for the security of the system might be forgotten by the time it is released to production.

These TODO comments should at least have a brief description of the task pending to do, and a link to the corresponding issue in the project repository.

Consider updating the TODO comments to add this information. For completeness and traceability, a signature and a timestamp can be added. For example:

```
// TODO: handle approval special cases like USDT, KNC, etc
// https://github.com/tallycash/tally-contracts/issues
// --mhluongo - 20210722
```

[N14] Indirect access of type extremae

In L310 and L311 of the `Tally` contract, the values `2 ** 256 - 1` and `2 ** 96 - 1` are used to represent max values of the respective types `uint256` and `uint96`.

As of Solidity v0.6.8, the max and min values for every integer type `T` can be accessed directly via the syntax `type(T).max` and `type(T).min`.

Use of this syntax can make the code more clear and readable to stakeholders, and can reduce bugs in implementation.

Consider accessing the extreme values of integer types directly using the `type(T).max`/`type(T).min` syntax.

[N15] Typos

The codebase contains the following typos:

- `each accounts delegate` should be `each account's delegate`.
- `contract mus be deployed` should be `contract must be deployed`.
- `caries` should be `carries`.
- In line 201 and 210 of the `HuntingGround` contract, `of greater` should be `or greater`.
- `getRewards()` should be `getReward()`.
- `uses` should be `using`.
- In line 131 of the `HuntingGround` contract, there's an additional "to" that should be removed.

Consider correcting these typos to improve code readability.

[N16] Unnecessary code

The `Tally` contract is deployed by the `TallyDeployer` in its constructor and it is requiring the `totalSupply` to be zero at construction time, but since any state variable is set to its default value if not initialized inline, there are no other ways in which `totalSupply` can be zero.

Any change applied to the contract that results in an initial `totalSupply` different from zero, would inherently produce a different bytecode and then a different address for the deployed contract. This would cause a revert in line 107 of the `TallyDeployer` contract.

Moreover, there is also an useless instruction to set the minter to the `address(0)` but this is also the default value.

Also, the global variables `periodFinish` and `rewardRate` of the `Unipool` contract are unnecessarily initialized to be 0.

Consider removing unnecessary checks and instructions to have a cleaner and more readable codebase, but also to avoid bugs and reduce attack surface in any future development.

[N17] Unused function

The `delegateTo` function of the `GovernorBravoDelegator` contract is not used in the codebase. This function was previously used, in the original Compound's codebase, to be called in the constructor. Since the constructor has been refactored, this function can be removed.

In order to improve clarity and quality of the codebase, consider removing the `delegateTo` function.

[N18] Unused library function

The `Math` library contains the following functions which are not used anywhere in the codebase

- subOrZero(uint128 a, uint128 b)
- subOrZero(uint64 a, uint64 b)
- subOrZero(uint32 a, uint32 b)

To improve the readability of the code, consider removing any unused library functions.

[N19] Unused struct

`Earning` struct is defined in the `HuntingGround` contract but is not used anywhere in the codebase.

To improve the readability of the code and reduce its size, consider removing the unused struct.

[N20] Unused function parameters

There were instances in this codebase where function parameters appear in a function signature but are never used within their respective function.

- The `account` parameter of the `_accountForStake` function.
- The `staker` parameter for both the `postStake` and the `preWithdraw` functions.

Consider removing these unused function parameters to avoid confusions.

[N21] Wrong visibility

The `calculateTotalPoolEarnings` function of the `UnipoolHuntingGround` contract is not changing the storage of the contract.

To silence compiler warnings, improve expliciteness and readability, consider adding the `view` visibiility to the function definition.

Conclusions

We are happy to see new designs being proposed into the space, however, six criticals and three high severity issues were found, among other issues which have lower severities. Recommendations have been proposed where possible, along with possible fixes. But where recommendations for patches were not straight-forward, general comments have been left. We strongly recommend to the team to fix the issues present in this report. Since such fixes will drastically change the code base, we also suggest going through another audit when the changes are done.