Moonwell
Findings & Analysis Report

2023-10-04

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Moonwell smart contract system written in Solidity. The audit took place between July 24—July 31 2023.

Wardens

78 Wardens contributed reports to the Moonwell:

1. immeas
2. T1MOH
3. kankodu
4. Udsen
5. 0xWaitress
6. said
7. ravikiranweb3
8. volodya
9. cryptonue
10. catellatech
11. hals
12. mert_eren

13. nadin
14. kutugu
15. Nyx
16. berlin-101
17. Sathish9098
18. Aymen0909
19. kodyvim
20. ast3ros
21. jaraxxus
22. sces60107
23. markus_ether
24. 0xComfyCat
25. ABAIKUNANBAEV
26. 0xkazim
27. Jigsaw
28. Vagner
29. bin2chen
30. MohammedRizwan
31. niki
32. Kaysoft
33. solsaver
34. BRONZEDISC
35. Auditwolf
36. Hama
37. okolicodes
38. R-Nemes
39. dacian
40. Tendency
41. 0x70C9

42. RED-LOTUS-REACH (BlockChomper, DedOhWale, SaharDevep, reentrant, and escrow)
43. 0xSmartContract
44. K42
45. 0xAnah
46. ChrisTina
47. JP_Courses
48. josephdara
49. twcctop
50. LosPollosHermanos (jc1 and scaraven)
51. Arz
52. Jorgect
53. 33audits
54. cats
55. Rolezn
56. Stormreckson
57. fatherOfBlocks
58. souilos
59. 0xl51
60. Topmark
61. naman1778
62. wahedtalash77
63. jamshed
64. petrichor
65. 0xackermann
66. 0xArcturus
67. lanrebayode77
68. 2997ms
69. John_Femi

70. banpaleo5
71. albertwh1te
72. codetilda
73. eeshenggoh

This audit was judged by alcueca.

Final report assembled by PaperParachute.

Summary

The C4 analysis yielded an aggregated total of 17 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 17 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 56 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Moonwell repository, and is composed of 15 smart contracts written in the Solidity programming language and includes 3,569 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic

- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Medium Risk Findings (17)

[M-01] Borrowers can avoid liquidations, if ERC777 token is configured as an `emissionToken`

Submitted by Udsen

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MErc20.sol#L139-L142

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MToken.sol#L1002

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L1235-L1239

If a borrower is `undercollateralized` then he can be liquidated by a liquidator by calling the `MErc20.liquidateBorrow` function. `liquidateBorrow` function calls the `MToken.liquidateBorrowFresh` in its execution process. Inside the `liquidateBorrowFresh` function the `MToken.repayBorrowFresh` is called which verifies whether repayment of borrowed amount is allowed by calling the `Comptroller.repayBorrowAllowed` function. The `repayBorrowAllowed` function updates the `borrower eligible rewards` by calling the `MultiRewardDistributor.updateMarketBorrowIndexAndDisburseBorrowerRewards`.

The `MultiRewardDistributor.updateMarketBorrowIndexAndDisburseBorrowerRewards` function calls the `disburseBorrowerRewardsInternal` to ditribute the multi

rewards to the `borrower`. The `disburseBorrowerRewardsInternal` calls the `sendReward` function if the `_sendTokens` flag is set to `true`.

`sendReward` function is called for each `emissionConfig.config.emissionToken` token of the `MarketEmissionConfig[]` array of the given `_mToken` market.

`sendReward` function transafers the rewards to the `borrower` using the `safeTransfer` function as shown below:

```
            token.safeTransfer(_user, _amount);
```

Problem here is that `emissionToken` can be `ERC777` token (which is backward compatible with ERC20) thus allowing a `malicious borrower` (the `recipient contract` of rewards in this case) to implement `tokensReceived` hook in its contract and revert the transaction inside the hook. This will revert the entire `liquidation` transaction. Hence the `undercollateralized borrower` can avoid the liquidation thus putting both depositors and protocol in danger.

Proof of Concept

```
        function liquidateBorrow(address borrower, uint repayAmount,
            (uint err,) = liquidateBorrowInternal(borrower, repayAmc
            return err;
        }
```

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MErc20.sol#L139-L142

```
            (uint repayBorrowError, uint actualRepayAmount) = repayE
```

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MToken.sol#L1002

```
            if (_amount > 0 && _amount <= currentTokenHoldings) {
```

```
            // Ensure we use SafeERC20 to revert even if the rev
            token.safeTransfer(_user, _amount);
            return 0;
        } else {
```

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L1235-L1239

Tools Used

VSCode

Recommended Mitigation Steps

Hence it is recommended to disallow any ERC777 tokens being configured as `emissionToken` in any `MToken` market and only allow ERC20 tokens as `emissionTokens`.

sorrynotsorry (Lookout) commented:

> This logic is a bit off as the malicious user will not be having the ERC777 MToken rewards anyway, so the value extracted from this scenario doesn't match.

> Only allow ERC20 tokens as emissionTokens

> Above mitigation recommendation is also matching the start of the submission. Erc777's are Erc20 tokens as well.

> **alcueca (Judge) decreased severity to Medium and commented**: The finding shows that rewards are distributed as part of a liquidation.
>
> If an ERC777 token is set as an emissionsToken, which has transfer hooks, a malicious borrower can revert on receiving rewards, and avoid liquidation. Avoiding liquidation in certain situations accrues bad debt for the protocol, or can be compounded with other vulnerabilities. This would be a valid Medium.

> @ElliotFriedman, do you have in your documentation anything mentioning that ERC777 tokens should not be used as emission tokens?

ElliotFriedman (Moonwell) confirmed and commented:

> This is a valid finding.

[M-02] Missing check for the max/min price in the `chainlinkOracle.sol` contract

Submitted by Oxkazim, *also found by* BRONZEDISC, markus_ether, Auditwolf, Hama, okolicodes, R-Nemes, kodyvim, nadin, MohammedRizwan, dacian, *and* niki

The `chainlinkOracle.sol` contract specially the `getChainlinkPrice` function using the aggregator v2 and v3 to get/call the `latestRoundData`. the function should check for the min and max amount return to prevent some case happen, something like this:

https://solodit.xyz/issues/missing-checks-for-chainlink-oracle-spearbit-connext-pdf

https://solodit.xyz/issues/m-16-chainlinkadapteroracle-will-return-the-wrong-price-for-asset-if-underlying-aggregator-hits-minanswer-sherlock-blueberry-blueberry-git

If a case like LUNA happens then the oracle will return the minimum price and not the crashed price.

Proof of Concept

The function `getChainlinkPrice`:

```
    function getChainlinkPrice(
            AggregatorV3Interface feed
        ) internal view returns (uint256) {
            (, int256 answer, , uint256 updatedAt, ) = AggregatorV3I
                .latestRoundData();
            require(answer > 0, "Chainlink price cannot be lower tha
            require(updatedAt != 0, "Round is in incompleted state")

            // Chainlink USD-denominated feeds store answers at 8 de
```

```
            uint256 decimalDelta = uint256(18).sub(feed.decimals());
            // Ensure that we don't multiply the result by 0
            if (decimalDelta > 0) {
                return uint256(answer).mul(10 ** decimalDelta);
            } else {
                return uint256(answer);
            }
        }
```

The function did not check for the min and max price.

Recommended Mitigation Steps

Some check like this can be added to avoid returning of the min price or the max price in case of the price crashes.

```
require(answer < _maxPrice, "Upper price bound breache
require(answer > _minPrice, "Lower price bound breache
```

sorrynotsorry (Lookout) commented:

> The implementation does not set a min/max value by design. Also Chainlink does not return min/max price as per the AggregatorV3 docs **HERE** contrary to the reported below;
>
> ChainlinkAggregators have minPrice and maxPrice circuit breakers built into them.
>
> Further proof required as per the context.

alcueca (Judge) commented:

> The warden is actually right. **It is a bit difficult to find**, but the `minAnswer` and `maxAnswer` can be retrieved from the Chainlink Aggregator, one step through the proxy. A circuit breaker should be implemented on the Moonwell oracle so that when the price edges close to `minAnswer` or `maxAnswer` it starts reverting, to avoid consuming stale prices when Chainlink freezes.

ElliotFriedman (Moonwell) acknowledged

[M-03] `excuteProposal` can fail due to Wormhole guardian change

Submitted by immeas

Wormhole governance can change signing guardian sets. If this happens between a proposal is queued and a proposal is executed. The second verification in `_executeProposal` will fail as the guardian set has changed between queuing and executing.

This would cause a proposal to fail to execute after the `proposalDelay` has passed. Causing a lengthy re-sending of the message from `Timelock` on source chain, then `proposalDelay` again.

Proof of Concept

To execute a proposal on `TemporalGovernor` it first needs to be queued. When queued it is checked that the message is valid against the Wormhole bridge contract:

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Governance/TemporalGovernor.sol#L295-L306

▶ Details

Were this to happen between `queueProposal` and `executeProposal` the proposal would fail to execute.

Recommended Mitigation Steps

Consider only check the `VAA` s validity against Wormhole in `_executeProposal` when it is fasttracked (`overrideDelay==true`).

However then anyone can just just take the hash from the proposal `VAA` and submit whatever commands. One idea to prevent this is to instead of storing the `VAA`

`vm.hash`, use the hash of the complete `VAA` as key in `queuedTransactions`. Thus, the content cannot be altered.

Or, the whole `VAA` can be stored alongside the timestamp and the `executeProposal` call can be made with just the hash.

ElliotFriedman (Moonwell) confirmed and commented:

> Good finding and this is expected behavior as wormhole guardians may change if signers are ever compromised.

[M-04] Malicious `emissionToken` could poison rewards for a market

Submitted by immeas, *also found by* mert_eren

When distributing rewards for a market, each `emissionConfig` is looped over and sent rewards for. `disburseBorrowerRewardsInternal` as an example, the same holds true for `disburseSupplierRewardsInternal`:

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L1147-L1247

```
File: MultiRewardDistributor/MultiRewardDistributor.sol

1147:     function disburseBorrowerRewardsInternal(
1148:         MToken _mToken,
1149:         address _borrower,
1150:         bool _sendTokens
1151:     ) internal {
1152:         MarketEmissionConfig[] storage configs = marketConf
1153:             address(_mToken)
1154:         ];

              // ... mToken balance and borrow index

1162:         // Iterate over all market configs and update their
```

```
1163:         for (uint256 index = 0; index < configs.length; inc
1164:             MarketEmissionConfig storage emissionConfig = c

                  // ... index updates

1192:             if (_sendTokens) {
1193:                 // Emit rewards for this token/pair
1194:                 uint256 pendingRewards = sendReward( // <--
1195:                     payable(_borrower),
1196:                     emissionConfig.borrowerRewardsAccrued[_
1197:                     emissionConfig.config.emissionToken
1198:                 );
1199:
1200:                 emissionConfig.borrowerRewardsAccrued[
1201:                     _borrower
1202:                 ] = pendingRewards;
1203:             }
1204:         }
1205:     }

...

1214:     function sendReward(
1215:         address payable _user,
1216:         uint256 _amount,
1217:         address _rewardToken
1218:     ) internal nonReentrant returns (uint256) {

              // ... short circuits breakers

1232:         uint256 currentTokenHoldings = token.balanceOf(addr
1233:
1234:         // Only transfer out if we have enough of a balance
1235:         if (_amount > 0 && _amount <= currentTokenHoldings)
1236:             // Ensure we use SafeERC20 to revert even if th
1237:             token.safeTransfer(_user, _amount); // <-- if t
1238:             return 0;
1239:         } else {
                  // .. default return _amount
1245:             return _amount;
1246:         }
1247:     }
```

If one transfer reverts the whole transaction fails and no rewards will be paid out for this user. Hence if there is a malicious token that would revert on transfer it would

cause no rewards to paid out. As long as there is some rewards accrued for the malicious `emissionConfig`. The users with already unclaimed rewards for this `emissionConfig` would have their rewards permanently locked.

The `admin` (`TemporalGovernor`) of `MultiRewardsDistributor` could update the reward speed for the token to `0` but that would just prevent further damage from being done.

Upgradeable tokens aren't unusual and hence the token might seem harmless to begin with but be upgraded to a malicious implementation that reverts on transfer.

Proof of Concept

Test in `MultiRewardDistributor.t.sol`, `MultiRewardSupplySideDistributorUnitTest`, most of the test is copied from `testSupplierHappyPath` with the addition of `MaliciousToken`:

```
    contract MaliciousToken {
        function balanceOf(address) public pure returns(uint256) {
            return type(uint256).max;
        }

        function transfer(address, uint256) public pure {
            revert("No transfer for you");
        }
    }



        function testAddMaliciousEmissionToken() public {
            uint256 startTime = 1678340000;
            vm.warp(startTime);

            MultiRewardDistributor distributor = createDistributorWi

            // malicious token added
            MaliciousToken token = new MaliciousToken();
            distributor._addEmissionConfig(
                mToken,
                address(this),
                address(token),
                1e18,
```

```
            1e18,
            block.timestamp + 365 days
        );

        comptroller._setRewardDistributor(distributor);

        emissionToken.allocateTo(address(distributor), 10000e18)

        vm.warp(block.timestamp + 1);

        mToken.mint(2e18);
        assertEq(MTokenInterface(mToken).totalSupply(), 2e18);

        // Wait 12345 seconds after depositing
        vm.warp(block.timestamp + 12345);

        // claim fails as the malicious token reverts on transfe
        vm.expectRevert("No transfer for you");
        comptroller.claimReward();

        // no rewards handed out
        assertEq(emissionToken.balanceOf(address(this)), 0);
    }
```

Recommended Mitigation Steps

Consider adding a way for `admin` to remove an `emissionConfig`.

Alternatively, the reward transfer could be wrapped in a `try` / `catch` and returning `_amount` in `catch`. Be mindful to only allow a certain amount of gas to the transfer then as otherwise the same attack works with consuming all gas available.

ElliotFriedman (Moonwell) disputed and commented:

> Emission creator (comptroller admin) and emission owners are trusted, it is assumed they will not add any poison reward tokens.

alcueca (Judge) commented:

> @ElliotFriedman, I'm not sure yet if anyone else has reported this, but the emissions token doesn't need to be even suspicious. USDC and USDT can

> blacklist users. If you use those tokens as emissions and **any** of your rewards holders get blacklisted, this issue will get triggered.

lyoungblood (Warden) commented:

> Since tokens (EmissionConfig) cannot be added except by admin (the DAO), I still dispute the validity of the finding, or at least the severity. We have to be reasonable in our assumptions and the assumption that the admin will add a malicious/censorable token can't really be a precursor to a valid finding. The admin can directly remove funds from the contract with removeReserves, but we wouldn't consider a finding like that valid.

alcueca (Judge) commented:

> @lyoungblood, USDC and USDT are both censorable, and it sounds pretty reasonable that would be used as emission tokens.

[M-05] Only `guardian` can change `guardian`

Submitted by **immeas**

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Governance/TemporalGovernor.sol#L27

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol#L81-L86

`guardian` is mentioned as an area of concern in the **docs**:

> Specific areas of concern include:
>
> - TemporalGovernor which is the cross chain governance contract. Specific areas of concern include delays, **the pause guardian**, ...

`guardian` is a roll that has the ability to pause and unpause `TemporalGovernor`. In code, it uses the `owner` from OpenZeppelin `Ownable` as `guardian`. The issue is that `Ownable::transferOwnership` is not overridden. Only `guardian` (`owner`) can transfer the role.

This can be a conflict of interest if there is a falling out between governance and the guardian. If the `guardian` doesn't want to abstain, governance only option would be to call `revokeGuardian` which sets `owner` to `address(0)`. This permanently removes the ability to pause the contract which can be undesirable.

Proof of Concept

Simple test in `TemporalGovernorExec.t.sol`:

```
function testGovernanceCannotTransferGuardian() public {
    address[] memory targets = new address[](1);
    targets[0] = address(governor);
    uint256[] memory values = new uint256[](1);

    bytes[] memory payloads = new bytes[](1);
    payloads[0] = abi.encodeWithSelector(Ownable.transferOwr

    bytes memory payload = abi.encode(address(governor), tar
    mockCore.setStorage(true, trustedChainid, governor.addre

    governor.queueProposal("");

    vm.warp(block.timestamp + proposalDelay);

    // governance cannot transfer guardian
    vm.expectRevert(abi.encodeWithSignature("Error(string)",
    governor.executeProposal("");
}
```

Recommended Mitigation Steps

Consider overriding `transferOwnership` and either limit it to only governance (`msg.sender == address(this)`) or both `guardian` and governance.

ElliotFriedman (Moonwell) confirmed and commented:

> This is an interesting finding. Only `guardian` can change `guardian`, however, `guardian` can only pause once and is limited in abilities to being able to fast track execution, and unpause. After a single malicious pause, the `guardian` would no longer be able to pause, and 30 days later, governance would reopen.

[M-06] `TemporalGovernor` can be bricked by `guardian`

Submitted by immeas, *also found by* markus_ether, 0xkazim, kutugu, T1MOH, bin2chen, *berlin-101 (*1, 2*), and* ABAIKUNANBAEV

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Governance/TemporalGovernor.sol#L27

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Governance/TemporalGovernor.sol#L188-L198

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Governance/TemporalGovernor.sol#L256

When a `guardian` pauses `TemporalGovernor` they lose the ability to pause again. This can then be reinstated by governance using `grantGuardiansPause`.

However `grantGuardiansPause` can be called when the contract is still paused. This would break the assertions in `permissionlessUnpause` and `togglePause`: `assert(!guardianPauseAllowed)`.

Also, `TemporalGovernor` inherits from OpenZeppelin `Ownable`. There the `owner` has the ability to `renounceOwnership`.

This could cause `TemporalGovernor` to end up in a bad state, since `revokeGuardian` call that has special logic for revoking the guardian.

There is the ability to combine these two issues to brick the contract:

1. `guardian` pauses the contract.
2. governance sends an instruction for `grantGuardiansPause` which the `guardian` executes (while still paused). This step disables both `permissionlessUnpause` and `togglePause`. Though the contract is still rescuable through `revokeGuardian`.

3. `guardian` calls `renounceOwnership` from `Ownable`. Which disables any possibility to execute commands on `TemporalGovernor`. Hence no more ability to call `revokeGuardian`.

Impact

The contract is bricked. Since `permissionlessUnpause` can't be called due to the `assert(!guardianPauseAllowed)`. No cross chain calls an be processed because `executeProposal` is `whenNotPaused` and there is no `guardian` to execute `fastTrackProposalExecution`. Thus the `TemporalGovernor` will be stuck in paused state.

It is also possible for the `guardian` to hold the contract hostage before step 3 (`renounceOwnership`) as they are the only ones who can execute calls on it (through `fastTrackProposalExecution`).

This is scenario relies on governance sending a cross chain `grantGuardiansPause` while the contract is still paused and a malicious `guardian` which together are unlikely. Though the docs mention this as a concern:

> Specific areas of concern include: ...
>
> - TemporalGovernor which is the cross chain governance contract. Specific areas of concern include delays, the pause guardian, and putting the contract into a state where it cannot be updated.

Proof of Concept

Test in `TemporalGovernorExec.t.sol`:

```
function testBrickTemporalGovernor() public {
    // 1. guardian pauses contract
    governor.togglePause();
    assertTrue(governor.paused());
    assertFalse(governor.guardianPauseAllowed());

    // 2. grantGuardianPause is called
    address[] memory targets = new address[](1);
    targets[0] = address(governor);
```

```
        uint256[] memory values = new uint256[](1);

        bytes[] memory payloads = new bytes[](1);
        payloads[0] = abi.encodeWithSelector(TemporalGovernor.gr

        bytes memory payload = abi.encode(address(governor), tar
        mockCore.setStorage(true, trustedChainid, governor.addre

        governor.fastTrackProposalExecution("");
        assertTrue(governor.guardianPauseAllowed());

        // 3. guardian revokesOwnership
        governor.renounceOwnership();

        // TemporalGovernor is bricked

        // contract is paused so no proposals can be sent
        vm.expectRevert("Pausable: paused");
        governor.executeProposal("");

        // guardian renounced so no fast track execution or togg
        assertEq(address(0),governor.owner());

        // permissionlessUnpause impossible because of assert
        vm.warp(block.timestamp + permissionlessUnpauseTime + 1)
        vm.expectRevert();
        governor.permissionlessUnpause();
    }
```

Recommended Mitigation Steps

However unlikely this is to happen, there are some simple steps that can be taken to prevent if from being possible:

Consider adding `whenNotPaused` to `grantGuardiansPause` to prevent mistakes (or possible abuse). And also overriding the calls `renounceOwnership` and `transferOwnership`. `transferOwnership` should be a governance call (`msg.sender == address(this)`) to move the `guardian` to a new trusted account.

Perhaps also consider if the `assert(!guardianPauseAllowed)` is worth just for pleasing the SMT solver.

ElliotFriedman (Moonwell) confirmed

[M-07] `fastTrackProposalExecution` doesn't check `intendedRecipient`

Submitted by immeas, *also found by* 0xComfyCat, Vagner, markus_ether, kutugu, ast3ros, *and* volodya

Wormhole cross chain communication is multicasted to all their supported chains:

> Please notice that there is no destination address or chain in these functions. VAAs simply attest that "this contract on this chain said this thing." Therefore, VAAs are multicast by default and will be verified as authentic on any chain they are brought to.

The `TermporalGovernor` contract handles this by checking `queueProposal` that the `intendedRecipient` is itself:

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Governance/TemporalGovernor.sol#L320-L322

```
File: core/Governance/TemporalGovernor.sol

320:         // Very important to check to make sure that the VAA
321:         // to be sent to this contract
322:         require(intendedRecipient == address(this), "Tempora
```

There is also a `fastTrackProposalExecution` that the guardian can perform (`owner` is referred ot as `guardian`):

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Governance/TemporalGovernor.sol#L261-L268

```
File: core/Governance/TemporalGovernor.sol

261:     /// @notice Allow the guardian to process a VAA when the
```

```
262:     /// Temporal Governor is paused this is only for use dur
263:     /// periods of emergency when the governance on moonbeam
264:     /// compromised and we need to stop additional proposals
265:     /// @param VAA The signed Verified Action Approval to pr
266:     function fastTrackProposalExecution(bytes memory VAA) ex
267:         _executeProposal(VAA, true); /// override timestamp
268:     }
```

This will bypass the `queueProposal` flow and execute commands immediately.

The issue here is that there is no check in `_executeProposal` that the `intendedRecipient` is this `TemporalGovernor`.

Impact

`governor` can execute any commands communicated across `Wormhole` as long as they are from a trusted source.

This requires that the `targets` line up between chains for a `governor` to abuse this. However only one `target` must line up as the `.call` made **does not check for contract existence**. Since "unaligned" addresses between chains will most likely not exist on other chains these calls will succeeed.

Proof of Concept

Test in `TemporalGovernorExec.t.sol`, all of the test is copied from `testProposeFailsIncorrectDestination` with just the change at the end where instead of calling `queueProposal`, `fastTrackProposalExecution` is called instead, which doesn't revert, thus highligting the issue.

```
    function testFasttrackExecuteSucceedsIncorrectDestination()
        address[] memory targets = new address[](1);
        targets[0] = address(governor);

        uint256[] memory values = new uint256[](1);
        values[0] = 0;

        TemporalGovernor.TrustedSender[]
            memory trustedSenders = new TemporalGovernor.Trustec
```

```
        trustedSenders[0] = ITemporalGovernor.TrustedSender({
            chainId: trustedChainid,
            addr: newAdmin
        });

        bytes[] memory payloads = new bytes[](1);

        payloads[0] = abi.encodeWithSignature( /// if issues use
            "setTrustedSenders((uint16,address)[])",
            trustedSenders
        );

        /// wrong intendedRecipient
        bytes memory payload = abi.encode(newAdmin, targets, val

        mockCore.setStorage(
            true,
            trustedChainid,
            governor.addressToBytes(admin),
            "reeeeeee",
            payload
        );

        // guardian can fast track execute calls intended for ot
        governor.fastTrackProposalExecution("");
    }
```

Recommended Mitigation Steps

Consider adding a check in `_executeProposal` for `intendedRecipient` same as is done in `_queueProposal`.

ElliotFriedman (Moonwell) confirmed and commented:

> This is a valid finding. However, in order for this to be exploitable, the proper governor would have to emit the proper calldata on another chain, with the exactly needed calldata formatting, and call publishMessage for this to be exploitable which is a 0% chance of happening.

[M-08] `fastTrackProposalExecution` should only be callable when `TemporalGovernor` is paused

Submitted by Aymen0909*, also found by* hals*,* sces60107*,* Jigsaw*, and* ABAIKUNANBAEV

The function `fastTrackProposalExecution` is supposed to be used by the `TemporalGovernor` owner during periods of emergency, when the contract is paused to fast track a certain proposal (queue and execute a proposal directly without delay or voting), with gaol of ensuring the protocol safety.

But the function `fastTrackProposalExecution` is missing the `whenPaused` modifier which means it can be called at any time to fast track any proposal and not just when the `TemporalGovernor` is paused.

I submit this issue as only Medium risk because only the `TemporalGovernor` owner can call the `fastTrackProposalExecution` function (and i suppose that the owner is trusted and will not execute malicious actions), but this issue still goes against the intent of the governance process in which the members of the DAO are supposed to propose, vote and execute the proposals in a permissionless and decentralized manner and where the owner should not be able to execute actions on the protocol without the agreement of the members (except for those emergency situation of course).

Proof of Concept

The issue is present in the `fastTrackProposalExecution` function below :

```
    /// @notice Allow the guardian to process a VAA when the
    /// Temporal Governor is paused this is only for use during
    /// periods of emergency when the governance on moonbeam is
    /// compromised and we need to stop additional proposals from go
    /// @param VAA The signed Verified Action Approval to process
    function fastTrackProposalExecution(bytes memory VAA) external {
        // @audit can be called when contract is not paused
        _executeProposal(VAA, true); /// override timestamp checks a
    }
```

As it can be seen the function does not contain the `whenPaused` modifier and thus it can be called at any moment by the owner to fast track a proposal.

We can also notice that the function comments do mention the fact that it should only be called when the Temporal Governor is paused.

Recommended Mitigation Steps

I recommend to add the `whenPaused` modifier to the `fastTrackProposalExecution` function, in order to ensure that the governance process will always work as a real DAO and the owner only intervene in the emergency cases.

ElliotFriedman (Moonwell) confirmed in a duplicate issue (245)

[M-09] Proposals which intend to send native tokens to target addresses can't be executed

Submitted by hals, *also found by* immeas, 0xComfyCat, RED-LOTUS-REACH, Kaysoft, Tendency, Vagner, 0xl51, sces60107, kodyvim, 0x70C9, T1MOH, *and* bin2chen

https://github.com/code-423n4/2023-07-moonwell/blob/fced18035107a345c31c9a9497d0da09105df4df/src/core/Governance/TemporalGovernor.sol#L237-L239

https://github.com/code-423n4/2023-07-moonwell/blob/fced18035107a345c31c9a9497d0da09105df4df/src/core/Governance/TemporalGovernor.sol#L400-L402

- In `TemporalGovernor` contract: any verified action approval (VAA)/proposal can be executed by anyone if it has been queued and passed the time delay.
- But if the proposal is intended to send native tokens to the target address; it will revert since `TemporalGovernor` contract doesn't have any balance, as there's no `receive()` or payable functions to receive the funds that will be sent to the proposal's target address.

- So any proposal with a value (decoded from `vm.payload`) will not be executed since the `target.call{value:value}(data)` will revert.

Proof of Concept

- Code: Line 237-239

```
File: src/core/Governance/TemporalGovernor.sol
Line 237-239:
    function executeProposal(bytes memory VAA) public whenNotPau
        _executeProposal(VAA, false);
    }
```

Line 400-402

```
File: src/core/Governance/TemporalGovernor.sol
Line 400-402:
        (bool success, bytes memory returnData) = target.call{va
            data
        );
```

- Foundry PoC:
- This test is copied from `testExecuteSucceeds` test in `TemporalGovernorExec.t.sol` file,and modified to demonstrate the issue; where a proposal is set to send an EOA receiverAddress a value of 1 ether, but will revert due to lack of funds (follow the comments in the test):

```
function testProposalWithValueExecutionFails() public {
        address receiverAddress = address(0x2);
        address[] memory targets = new address[](1);
        targets[0] = address(receiverAddress);

        uint256[] memory values = new uint256[](1);
        values[0] = 1 ether; // the proposal has a value of 1 et

        bytes[] memory payloads = new bytes[](1);
```

```
        payloads[0] = abi.encodeWithSignature("");

        /// to be unbundled by the temporal governor
        bytes memory payload = abi.encode(
            address(governor),
            targets,
            values,
            payloads
        );

        mockCore.setStorage(
            true,
            trustedChainid,
            governor.addressToBytes(admin),
            "reeeeeee",
            payload
        );
        governor.queueProposal("");

        bytes32 hash = keccak256(abi.encodePacked(""));
        (bool executed, uint248 queueTime) = governor.queuedTrar

        assertEq(queueTime, block.timestamp);
        assertFalse(executed);
        //---executing the reposal
        vm.warp(block.timestamp + proposalDelay);

        //check that the balance of receiverAddress is zero befc
        assertEq(receiverAddress.balance, 0);

        // executeProposal function will revert due to lack of f
        vm.expectRevert();
        governor.executeProposal("");
        assertFalse(executed); // the proposal wasn't executed
        assertEq(receiverAddress.balance, 0); // the receiverAdc
    }
```

2. Test result:

```
$ forge test --match-test testProposalWithValueExecutionFails -v
Running 1 test for test/unit/TemporalGovernor/TemporalGovernorEx
[PASS] testProposalWithValueExecutionFails() (gas: 458086)
Test result: ok. 1 passed; 0 failed; finished in 2.05ms
```

Tools Used

Foundry.

Recommended Mitigation Steps

Add `receive()` function to the `TemporalGovernor` contract; so that it can receive funds (native tokens) to be sent with proposals.

ElliotFriedman (Moonwell) confirmed and commented:

> Good finding!

[M-10] Initial deploy won't succeed because of too high `initialMintAmount` for USDC market

Submitted by **T1MOH**, *also found by* **immeas**

At the time of deploy, deployer initializes token markets with initial amount to prevent exploit. At least USDC and WETH markets will be initialized during deploy. But `initialMintAmount` is hardcoded to `1e18` which is of for WETH (1,876 USD), but unrealistic for USDC (1,000,000,000,000 USD). Therefore deploy will fail.

Proof of Concept

Here `initialMintAmount = 1 ether`:
https://github.com/code-423n4/2023-07-moonwell/blob/fced18035107a345c31c9a9497d0da09105df4df/test/proposals/Configs.sol#L55

```
        /// @notice initial mToken mint amount
        uint256 public constant initialMintAmount = 1 ether;
```

Here this amount is approved to MToken contract and supplied to mint MToken:
https://github.com/code-423n4/2023-07-moonwell/blob/fced18035107a345c31c9a9497d0da09105df4df/test/proposals/mips/mip00.sol#L334-L379

```
        for (uint256 i = 0; i < cTokenConfigs.length; i++) {
            Configs.CTokenConfiguration memory config = cTok

            address cTokenAddress = addresses.getAddress(
                config.addressesString
            );

            ...

            /// Approvals
            _pushCrossChainAction(
                config.tokenAddress,
                abi.encodeWithSignature(
                    "approve(address,uint256)",
                    cTokenAddress,
                    initialMintAmount
                ),
                "Approve underlying token to be spent by mar
            );

            /// Initialize markets
            _pushCrossChainAction(
                cTokenAddress,
                abi.encodeWithSignature("mint(uint256)", ini
                "Initialize token market to prevent exploit'
            );

            ...
        }
```

Recommended Mitigation Steps

Specify `initialMintAmount` for every token separately in config

ElliotFriedman (Moonwell) confirmed, but disagreed with severity and commented:

> Valid issue, but we have already fixed this. Probably a low severity issue as this deploy script was only configured to work on local testnets where we control the ERC20 tokens.

alcueca (Judge) commented:

> The issue is still valid as Medium with the code and knowledge available to the wardens.

[M-11] Incorrect address is set as Wormhole Bridge, which breaks deploy

Submitted by T1MOH

Wormhole Bridge will have incorrect address, which disables whole TemporalGovernance contract and therefore administration of Moonbeam. But during deployment markets are initialized with initial token amounts to prevent exploits, therefore TemporalGovernance must own this initial tokens. But because of incorrect bridge address TemporalGovernance can't perform any action and these tokens are brick. Protocol needs redeployment and loses initial tokens.

Proof of Concept

To grab this report easily I divided it into 3 parts

1. Why `WORMHOLE_CORE` set incorrectly

There is no config for Base Mainnet, however this comment states for used parameters:
https://github.com/code-423n4/2023-07-moonwell/blob/fced18035107a345c31c9a9497d0da09105df4df/test/proposals/mips/mip00.sol#L55-L61

```
/// ------------------------------------------------------
/// Chain Name              Wormhole Chain ID   Network ID
///  Ethereum (Goerli)         2                    5
///  Ethereum (Sepolia)        10002           11155111
///  Base                     30              84531       0xA3
///  Moonbeam                    16               1284
/// ------------------------------------------------------
```

Why to treat this comment? Other parameters are correct except Base Network ID. But Base Network ID has reflection in code, described in #114. 0xA31aa3FDb7aF7Db93d18DDA4e19F811342EDF780 is address of Wormhole Token

Bridge on Base Testnet [according to docs] and this address has no code [in Base Mainnet].

2. Why governor can't execute proposals with incorrect address of Wormhole Bridge

Proposal execution will revert here because address `wormholeBridge` doesn't contain code:
https://github.com/code-423n4/2023-07-moonwell/blob/fced18035107a345c31c9a9497d0da09105df4df/src/core/Governance/TemporalGovernor.sol#L344-L350

```
function _executeProposal(bytes memory VAA, bool overrideDel
    // This call accepts single VAAs and headless VAAs
    (
        IWormhole.VM memory vm,
        bool valid,
        string memory reason
    ) = wormholeBridge.parseAndVerifyVM(VAA);
    ...
}
```

3. Impact of inability to execute proposals in TemporalGovernor.sol

Here you can see that deployer should provide initial amounts of tokens to initialize markets:

```
/// @notice the deployer should have both USDC, WETH and any
/// listed to be able to deploy on base. This allows the dep
/// markets with a balance to avoid exploits
function deploy(Addresses addresses, address) public {
...
}
```

And here governor approves underlying token to MToken and mints initial mTokens. It means that governor has tokens on balance.

```
function build(Addresses addresses) public {
```

```
/// Unitroller configuration
_pushCrossChainAction(
    addresses.getAddress("UNITROLLER"),
    abi.encodeWithSignature("_acceptAdmin()"),
    "Temporal governor accepts admin on Unitroller"
);

...

/// set mint unpaused for all of the deployed MTokens
unchecked {
    for (uint256 i = 0; i < cTokenConfigs.length; i++) {
        Configs.CTokenConfiguration memory config = cTok

        address cTokenAddress = addresses.getAddress(
            config.addressesString
        );

        _pushCrossChainAction(
            unitrollerAddress,
            abi.encodeWithSignature(
                "_setMintPaused(address,bool)",
                cTokenAddress,
                false
            ),
            "Unpause MToken market"
        );

        /// Approvals
        _pushCrossChainAction(
            config.tokenAddress,
            abi.encodeWithSignature(
                "approve(address,uint256)",
                cTokenAddress,
                initialMintAmount
            ),
            "Approve underlying token to be spent by mar
        );

        /// Initialize markets
        _pushCrossChainAction(
            cTokenAddress,
            abi.encodeWithSignature("mint(uint256)", ini
            "Initialize token market to prevent exploit'
        );
```

```
                ...
            }
        }

        ...
    }
```

To conclude, there is no way to return the tokens from the TemporalGovernor intended for market initialization

Recommended Mitigation Steps

Change `WORMHOLE_CORE` to 0xbebdb6C8ddC678FfA9f8748f85C815C556Dd8ac6 [according to docs]

[ElliotFriedman (Moonwell) disputed and commented]:

> This is correct on base goerli, however we have not added base mainnet contract yet as it has not been deployed.

> This can't be a high or even medium risk bug because this is an issue in our testnet deploy configuration.

[alcueca (Judge) decreased severity to Medium and commented]:

> Upon discussion with the sponsor about differences between this issue and #114, and actual impact of this issue, I'm instating Medium severity.

[M-12] Incorrect chainId of Base in deploy script will force redeployment

Submitted by [T1MOH]

Incorrect chainId of Base in deploy parameters results in incorrect deploy and subsequent redeployment.

Proof of Concept

Contract ChainIds.sol is responsible for mapping `chainId -> wormholeChainId` which is used in contract `Addresses` to associate contract name with its address on specific chain. `Addresses` is the main contract which keeps track of all dependency addresses and passed into main `deploy()` and here addresses accessed via block.chainId:
https://github.com/code-423n4/2023-07-moonwell/blob/fced18035107a345c31c9a9497d0da09105df4df/test/proposals/mips/mip00.sol#L77

```
function deploy(Addresses addresses, address) public {
    ...
        trustedSenders[0].chainId = chainIdToWormHoleId[bloc

    ...
        memory cTokenConfigs = getCTokenConfigurations(block
}
```

Here you can see that Network ID of Base set to 84531. But actual network id is 8453 from Base docs.

```
contract ChainIds {
    uint256 public constant baseChainId = 84531;
    uint16 public constant baseWormholeChainId = 30; /// TODO up

    uint256 public constant baseGoerliChainId = 84531;
    uint16 public constant baseGoerliWormholeChainId = 30;

    ...

    constructor() {
        ...
        chainIdToWormHoleId[baseChainId] = moonBeamWormholeChair
        ...
    }
}
```

Recommended Mitigation Steps

Change Base Network ID to 8453.

ElliotFriedman (Moonwell) confirmed

[M-13] It's not possible to liquidate deprecated market

Submitted by volodya, *also found by* Udsen

Currently in the code that is a function `_setBorrowPaused` that paused pause borrowing. In origin compound code `borrowGuardianPaused` is used to do liquidate markets that are bad. So now there is not way to get rid of bad markets.

```
function liquidateBorrowAllowed(
    address mTokenBorrowed,
    address mTokenCollateral,
    address liquidator,
    address borrower,
    uint repayAmount) override external view returns (uint)
    // Shh - currently unused
    liquidator;

    if (!markets[mTokenBorrowed].isListed || !markets[mToker
        return uint(Error.MARKET_NOT_LISTED);
    }

    /* The borrower must have shortfall in order to be liqui
    (Error err, , uint shortfall) = getAccountLiquidityInter
    if (err != Error.NO_ERROR) {
        return uint(err);
    }
    if (shortfall == 0) {
        return uint(Error.INSUFFICIENT_SHORTFALL);
    }

    /* The liquidator may not repay more than what is allowe
    uint borrowBalance = MToken(mTokenBorrowed).borrowBalanc
    uint maxClose = mul_ScalarTruncate(Exp({mantissa: closeI
    if (repayAmount > maxClose) {
        return uint(Error.TOO_MUCH_REPAY);
    }

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

src/core/Comptroller.sol#L394

Here is a list why liquidating deprecated markets can be necessary:

1. Deprecated markets may pose security risks, especially if they are no longer actively monitored or maintained. By liquidating these markets, the platform reduces the potential for vulnerabilities and ensures that user funds are not exposed to unnecessary risks.
2. Deprecated markets might require ongoing resources to maintain and support, including development efforts and infrastructure costs. By liquidating these markets, the platform can optimize resources and focus on more actively used markets and features.
3. Older markets might be based on legacy smart contracts that lack the latest security enhancements and improvements. Liquidating these markets allows the platform to migrate users to more secure and up-to-date contracts.
4. Deprecated markets may result in a fragmented user base, leading to reduced liquidity and trading activity. By consolidating users onto actively used markets, the platform can foster higher liquidity and better user engagement.

Recommended Mitigation Steps

I think compound have a way to liquidate deprecated markets for a safety reason, so it needs to be restored.

```
        function liquidateBorrowAllowed(
            address mTokenBorrowed,
            address mTokenCollateral,
            address liquidator,
            address borrower,
            uint repayAmount) override external view returns (uint)
            // Shh - currently unused
            liquidator;
            /* allow accounts to be liquidated if the market is depr
    +        if (isDeprecated(CToken(cTokenBorrowed))) {
    +            require(borrowBalance >= repayAmount, "Can not repa
    +            return uint(Error.NO_ERROR);
    +        }
            if (!markets[mTokenBorrowed].isListed || !markets[mToker
                return uint(Error.MARKET_NOT_LISTED);
            }
```

```
            /* The borrower must have shortfall in order to be liqui
            (Error err, , uint shortfall) = getAccountLiquidityInter
            if (err != Error.NO_ERROR) {
                return uint(err);
            }
            if (shortfall == 0) {
                return uint(Error.INSUFFICIENT_SHORTFALL);
            }

            /* The liquidator may not repay more than what is allowe
            uint borrowBalance = MToken(mTokenBorrowed).borrowBalanc
            uint maxClose = mul_ScalarTruncate(Exp({mantissa: closeF
            if (repayAmount > maxClose) {
                return uint(Error.TOO_MUCH_REPAY);
            }

            return uint(Error.NO_ERROR);
        }
+     function isDeprecated(CToken cToken) public view returns (b
+         return
+         markets[address(cToken)].collateralFactorMantissa == 0
+         borrowGuardianPaused[address(cToken)] == true &&
+         cToken.reserveFactorMantissa() == 1e18
+         ;
+     }
```

ElliotFriedman (Moonwell) confirmed and commented:

> This is a valid finding, nice find!

alcueca (Judge) commented:

> Despite the lack of PoC, this finding states clearly why the feature should be restored.

[M-14] Borrower and Supplier rewards accrued could be lost when Admin replaces the reward distributor with a new reward distributor

Submitted by ravikiranweb3

Comptroller's admin user can replace the rewards distributor contract with a new MultiRewardDistributor using _setRewardDistributor() function.

At that time, if the supplier and borrowers had accured rewards that are not distributed. The _setRewardDistributor() function replaces the old reward distributor with the new one without disbursing the borrower and supplier rewards accured.

The new logic of computation and distribution could be different in the newer version and hence before replacement, the accured rewards should be distributed using the existing logic.

The new accruals could take effect with nil accruals providing a clean slate.

Proof of Concept

The setRewardDistributor updates the mutiRewardDistributor contract with a new instance with out transferring the accured rewards to borrowers and lenders.

```
function _setRewardDistributor(MultiRewardDistributor newRew
    require(msg.sender == admin, "Unauthorized");
    MultiRewardDistributor oldRewardDistributor = rewardDist
    rewardDistributor = newRewardDistributor;
    emit NewRewardDistributor(oldRewardDistributor, newRewar
}
```

The above set function should be added with a logic to look at all the markets and for each market, the borrower and supplier rewards should be disbursed before the new instance is updated. As long as there are accured rewards, the old instance should not be replaced.

Recommended Mitigation Steps

Approach 1: Onchain approach

Step 1: Maintain a list of all accounts that participate in borrowing and supplying. getAllParticipants() returns a list of participants.

Step 2: Add an internal function disburseAllRewards() which can be called by adminOnly. This function looks for all partitipant accounts and for all markets, it

claims rewards before updating the rewards distributor with a new instance.

```
function disburseAllRewards() internal adminOnly {
        address[] memory holders = getAllParticipants();
        MToken[] memory mTokens = getAllMarkets();
        claimReward(holders, mTokens, true, true);
}

// @audit use the existing claimReward function to disburse the
```

Approach 2: Offchain approach

As the number of participants and markets grow, the onchain approach may result in DOS as the size of computations may not fit in a single block due to gas limit.

As an alternative, the claimReward() could be called from the outside for all the holders and markets in smaller chunks.

Use getOutstandingRewardsForUser() function to check if there are any un-disbursed rewards in the MultiRewarddistributor contract. Only if there are no un-disbursed rewards, then only allow the admin to replace the reward distributor contract with a new instance.

Taking these steps will prevent the potential case where borrowers and suppliers could loose accured rewards.

Key drive away with this approach is, unless all the accured are distributed and they is nothing to distribute, dont update the multi reward distributor contract.

ElliotFriedman (Moonwell) confirmed, but disagreed with severity and commented:

> This is valid, but disagree with severity since admin is trusted. Otherwise, if admin isn't trusted, then they could rug all the funds in the contract using the `_rescueFunds` method.

alcueca (Judge) commented:

> @ElliotFriedman - Note that the admin doesn't need to be malicious. You could be a few months down the line, and receive a bug report that forces you to pause and replace the rewards distributor. It would be really uncomfortable to move the not claimed rewards to the new contract.

[M-15] `accrueInterest` is expected to revert when the rate is higher than the maximum allowed rate, which is possible since the utilization can be more than 1

Submitted by 0xWaitress*, also found by* 0xWaitress*,* ast3ros*, catellatech (1, 2), Nyx (1, 2),* kodyvim*, and nadin (1, 2)*

accrueInterest is expected to revert when the rate is higher than the maximum allowed rate, which is possible since the utilization can be more than 1

accrueInterest is an essential function to keep updated of the global mToken interest payment from the borrower to the depositor. The function is called anytime there is a deposit/liquidate/borrow/repay/withdraw.

The function is set to revert when the borrowRateMantissa is bigger than the borrowRateMaxMantissa.

```
require(borrowRateMantissa <= borrowRateMaxMantissa, "borrow rate is absurdly high");.
```

With proper configuration, this check should be fine since the real borrowRate should be calibrated to be within the maxRate. However, since the utilization could actually be bigger than 1 (when reserve is bigger than cash [issue-37] , the actual rate could be bigger than the expected unreachable max rate:

Example:
Base: 1%
multiplierPerTimestamp: 5% APR
jumpMultiplier: 50% APR
klink: 50%
We could reasonably assume the max APR is 1% + 5% * 0.5 + 50% * (1-0.5) = 28.5%

However when reserve is more than cash it would cause the utilization be bigger than 1, let'say utilization is 101%: such that the actual rate would be:

1% + 5% * 0.5 + 50% * (1.01 - 0.5) = 29%

This would cause the accurueInterest to revert.

Impact: all deposit/withdraw/borrow/repay function would fail and severe brick the operation of the protocol and lock user funds. Interest accrual cannot work which impact both borrower for timely exit as well as deposit to collect their interest.

```
function accrueInterest() virtual override public returns (u
    /* Remember the initial block timestamp */
    uint currentBlockTimestamp = getBlockTimestamp();
    uint accrualBlockTimestampPrior = accrualBlockTimestamp;

    /* Short-circuit accumulating 0 interest */
    if (accrualBlockTimestampPrior == currentBlockTimestamp)
        return uint(Error.NO_ERROR);
    }

    /* Read the previous values out of storage */
    uint cashPrior = getCashPrior();
    uint borrowsPrior = totalBorrows;
    uint reservesPrior = totalReserves;
    uint borrowIndexPrior = borrowIndex;

    /* Calculate the current borrow interest rate */
    uint borrowRateMantissa = interestRateModel.getBorrowRat
    borrowsPrior, reservesPrior);
    require(borrowRateMantissa <= borrowRateMaxMantissa, "bo
    absurdly high");
```

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MToken.sol#L403

Utilization can be above 100% when reserve is more than cash

Note: per judge request, this information from duplicate issue 37 has been appended to M-16 for the final report.

Utilization is calculated as `borrow / (cash + borrow - reserve)` . and the utilization would determine the interest rate in this classic jump/klink model.

Consider:

1. both depositor and borrowers come in. Reserves accumulates as interest revenue paid by borrowers accrues.
2. depositor withdraws. The protocol has only borrowers and the reserve.

When reserves becomes more than cash, the formula for utilizationRate would give a utilization rate above 1. This is different from the code comment which said to

```
@return The utilization rate as a mantissa between [0, 1e18]
```

```
         * @return The utilization rate as a mantissa between [0, 1e
        function utilizationRate(uint cash, uint borrows, uint reser
            // Utilization rate is 0 when there are no borrows
            if (borrows == 0) {
                return 0;
            }

            return borrows.mul(1e18).div(cash.add(borrows).sub(reser
        }
```

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/IRModels/JumpRateModel.sol#L66-L75

Recommended Mitigation Steps

- Consider to cap the real-time borrowRate as the borrowRateMaxMantissa instead of reverting.
- Consider put a cap of 1e18 to the final output.

ElliotFriedman (Moonwell) disputed and commented:

> Reserves can be pulled back to admin address by admin, so this is a non issue.

[M-16] A single emissionCap is not suitable for different tokens reward if they have different underlying decimals

Submitted by OxWaitress

At MultiRewardDistributor, different rewards token can be set up for each mToken, which is the assetToken or debtToken of the lending protocol. When a reward token is set up, it's emission speed cannot be bigger than the emissionCap (100 * 1e18 as suggested in the code comment). However since multiple reward tokens may be set up, and they may have different decimal places, for example USDT has a decimal of 6 and GUSD even has a decimal of 2 they are virtually not bounded by such emissionCap.

On the contract, tokens with more than 18 decimal would not be practically emitted. This one-size-fit-all emissionCap might create obstacles for the protocol to effectively manage each rewardTokens.

```
/// @notice The emission cap dictates an upper limit for rew
/// @dev By default, is set to 100 1e18 token emissions / se
///  computation/multiplication overflows
uint256 public emissionCap;
```

```
function _updateBorrowSpeed(
    MToken _mToken,
    address _emissionToken,
    uint256 _newBorrowSpeed
) external onlyEmissionConfigOwnerOrAdmin(_mToken, _emissior
    MarketEmissionConfig
        storage emissionConfig = fetchConfigByEmissionToken(
            _mToken,
            _emissionToken
        );

    uint256 currentBorrowSpeed = emissionConfig
        .config
        .borrowEmissionsPerSec;

    require(
        _newBorrowSpeed != currentBorrowSpeed,
        "Can't set new borrow emissions to be equal to curre
    );
```

```
            require(
                _newBorrowSpeed < emissionCap,
                "Cannot set a borrow reward speed higher than the em
            );
```

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L703-L725

Recommended Mitigation Steps

Consider creating emissionCap for each rewardTokens so they can be adapted based on their own decimals.

ElliotFriedman (Moonwell) confirmed, but disagreed with severity and commented:

> This is a known issue, the amount of tokens created is bounded 100e18 per second. then, reward speeds will be integration tested to ensure they are properly configured.

alcueca (Judge) commented:

> @ElliotFriedman, I'm not sure if you understood the finding. How would you set up the emissions cap if emissions are in USDT (6 decimals) and WETH (18 decimals).

ElliotFriedman (Moonwell) commented:

> Valid issue, just a question of is it medium severity. I'll leave that for judges to decide.

alcueca (Judge) commented:

> Function incorrect as to spec would be QA - This means that if the way the code works is wrong but has no real impact in the business it is QA.

> Function of the protocol or its availability possibly impacted would be Medium - In certain cases, the emissionsCap will either not work or stop emissions, the protocol is impacted in a non-lethal manner, and the finding is medium.

lyoungblood (Warden) commented:

> Good finding - disagree with the severity as the cap is just a safety feature that didn't exist at all in the original Compound.

[M-17] borrowRateMaxMantissa should be specific to the chain protocol is being deployed to

Submitted by kankodu, *also found by* cryptonue

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MTokenInterfaces.sol#L23

https://github.com/code-423n4/2023-07-moonwell/blob/fced18035107a345c31c9a9497d0da09105df4df/src/core/MToken.sol#L403

- The purpose of borrowRateMaxMantissa is to put the protocol in failure mode when absurd utilisation makes borrowRate absurd. It is defined as a constant for all the chains. It should really be changed according to average blocktime of the chain the protocol is being deployed to.
- borrowRateMaxMantissa = 0.0005e16 translates to maximum borrow rate of .0005% / block.
- For Ethereum chain that has 12 seconds of average block time, this translates to maximum borrow rate of `0.0005% * (365 * 24 * 3600)/12 = 1314`. For BNB with 3 seconds of average block time it is `0.0005% * (365 * 24 * 3600)/3 = 5256%`. For fantom with 1 second of average block time it is `0.0005% * (365 * 24 * 3600)/1 = 15768%`.

Proof of Concept

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MTokenInterfaces.sol#L23

https://github.com/code-423n4/2023-07-moonwell/blob/fced18035107a345c31c9a9497d0da09105df4df/src/core/MToken.sol#L403

Recommended Mitigation Steps

- Decide on a maximum borrow rate protocol is ok with (my suggestion is ~1000%) and change `borrowRateMaxMantissa` according to what chain it is being deployed to.

ElliotFriedman (Moonwell) acknowledged and commented:

> Agreed this could be configured on a per chain basis, however this is legacy code that we don't want to change. This instance is only being deployed on base, so maybe instead of making it configurable, this value could be adjusted downwards.

Low Risk and Non-Critical Issues

For this audit, 56 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **immeas** received the top score from the judge.

The following wardens also submitted reports: 0xAnah, ChrisTina, JP_Courses, Udsen, cryptonue, 0xkazim, kutugu, Aymen0909, 0xComfyCat, jaraxxus, josephdara, Kaysoft, Vagner, hals, twcctop, said, LosPollosHermanos, ast3ros, Arz, Jorgect, 33audits, solsaver, Tendency, Nyx, cats, mert_eren, kodyvim, Sathish9098, MohammedRizwan, 0x70C9, Rolezn, catellatech, berlin-101, nadin, T1MOH, Stormreckson, fatherOfBlocks, ravikiranweb3, 0xWaitress, niki, souilos,kankodu, Topmark, naman1778, wahedtalash77, jamshed, petrichor, 0xackermann, 0xArcturus, lanrebayode77, 2997ms, John_Femi, banpaleo5, albertwh1te, codetilda, *and* eeshenggoh.

Low-risk findings (6)

[L-01] `guardian` can fast track every proposal

In `TemporalGovernor` the `guardian` has a special permission that they can `fastTrackProposalExecution` which will **bypass the normal** `queueProposal` **flow**:

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Governance/TemporalGovernor.sol#L261-L268

```
File: core/Governance/TemporalGovernor.sol

261:     /// @notice Allow the guardian to process a VAA when the
262:     /// Temporal Governor is paused this is only for use dur
263:     /// periods of emergency when the governance on moonbeam
264:     /// compromised and we need to stop additional proposals
265:     /// @param VAA The signed Verified Action Approval to pr
266:     function fastTrackProposalExecution(bytes memory VAA) ex
267:         _executeProposal(VAA, true); /// override timestamp
268:     }
```

This as the comment suggest should only be done in emergencies.

However there is nothing indicating that the `VAA` in question is supposed to be fast tracked. Hence a `guardian` can always fast track any `VAA` s that they please, bypassing the `queueProposal` functionality.

Recommendation

Consider adding another parameter to the `payload` indicating if the `VAA` is intended to be fast tracked or not.

[L-02] `TemporalGovernor` can get the same address on different chains

In `mip00.sol` the deploy and configuration proceedure for the `TemporalGovernor` is detailed:

https://github.com/code-423n4/2023-07-moonwell/blob/main/test/proposals/mips/mip00.sol#L103-L109

```
File: test/proposals/mips/mip00.sol

103:             /// this will be the governor for all the contra
104:             TemporalGovernor governor = new TemporalGovernor
105:                 addresses.getAddress("WORMHOLE_CORE"),
```

```
106:                 proposalDelay,
107:                 permissionlessUnpauseTime,
108:                 trustedSenders
109:             );
```

`new` will invoke the `CREATE` opcode. This only relies on the `msg.sender` and `nonce`. Since the `deploy` call executes a certain amount of calls when it executes the deploy of `TemporalGovernor` it will have certain `nonce`. Hence the address of `TemporalGovernor` will be dependent on the account that executes it. If this is run by a new account the address of `TemporalGovernor` can be the same if it is executed by the same new account on another chain.

This is important because if two `TemporalGovernor` contracts on different chains gets the same address they can execute each others proposals:

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Governance/TemporalGovernor.sol#L320-L322

```
File: core/Governance/TemporalGovernor.sol

320:         // Very important to check to make sure that the VAA
321:         // to be sent to this contract
322:         require(intendedRecipient == address(this), "Tempora
```

Recommendation

Pay attention to which accounts and how the execution of the deploy script is done. Perhaps use `CREATE2` with a random `salt` to deploy `TemporalGovernor` to guarantee it gets a different address on each chain.

[L-03] `TemporalGovernor::grantGuardiansPause` can be called after `guardian` is revoked

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Governance/TemporalGovernor.sol#L187-L198

```
File: Governance/TemporalGovernor.sol
```

```
187:     /// @notice grant the guardians the pause ability
188:     function grantGuardiansPause() external {
189:         require(
190:             msg.sender == address(this),
191:             "TemporalGovernor: Only this contract can update
192:         );
193:
194:         guardianPauseAllowed = true;
195:         lastPauseTime = 0;
196:
197:         emit GuardianPauseGranted(block.timestamp);
198:     }
```

This doesn't do any thing bad other than set `guardianPauseAllowed=true`, which is set to false in `revokeGuardian`.

Recommendation

Consider only allowing calls to `grantGuardiansPause` when `guardian != address(0)`.

[L-04] No cap on how many `emissionConfig`s there can be for a market

Too many `emissionConfig`s could lead to out of gas errors when updating reward indexes. Since every operation on an `mToken` triggers this the whole market would be DoSed if too many `emissionConfig`s would be added.

Recommendation

Consider adding a way to remove an `emissionConfig`

[L-05] Neither `_setMarketBorrowCaps` or `_setMarketSupplyCaps` checks if market is listed

When adding a cap on borrow/supply `admin` or `borrow/supplyCapGuardian` can call [`_setMarketBorrowCaps`])(https://github.com/code-423n4/2023-07-

moonwell/blob/main/src/core/Comptroller.sol#L807-L819) or `_setMarketSupplyCaps` respectively.

However in none of the there is any check that the market (`MToken`) for which they add the supply/borrow cap is actually listed. Hence `admin` or `guardian` could by mistake send the wrong address and then think that they configured the cap for that market when they didn't.

Recommendation

Add a check that the `MToken` is listed when setting borrow/supply caps.

[L-06] Lack of tests for `supplyCap`

Moonwell introduces a new feature to the `Comptroller` : `supplyCap` per market. This limits how much can be minted to that market. There is however no tests for this new feature. Test guarantee that the feature works as expected and will continue to work as expected when doing changes to the code. As such they are a core guarantee for code and protocol safety.

Consider adding a few tests for the `supplyCap` feature

Suggestions (1)

[S-01] `ChainlinkCompositeOracle::getDerivedPriceThreeOracles` is very specialized

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Oracles/ChainlinkCompositeOracle.sol#L157-L159

```
File: Oracles/ChainlinkCompositeOracle.sol

157:         return
158:             ((firstPrice * secondPrice * thirdPrice) / scali
159:                 .toUint256();
```

Here the three prices for the oracles are multiplied together. This requires a setup of prices like this: First one is stated to be `ETH / USD`

so there would have to be oracles:

`ETH / USD`, `A / ETH`, `B / A` to get the price `B / USD`

This is very limited which chainlink prices support "chaining" like this, the one used in integration test seems to one of the few:

`ETH / USD`, `stETH / ETH`, `wstETH / stETH` on Arbitrum.

Recommendations

Consider adding the ability to either multiply or divide by the last one, as that would give greater flexibility in which feeds can be used.

Refactoring (4)

[R-01] `MultiRewardDistributor::calculateNewIndex` parameter `_currentTimestamp` is confusing

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L906

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L914

```
File: MultiRewardDistributor/MultiRewardDistributor.sol

906:      * @param _currentTimestamp The current index timestamp

914:         uint32 _currentTimestamp,
```

`_currentTimestamp` implies *now* when it is however the last index timestamp. Consider renaming it to `lastIndexTimestamp`.

[R-02] Unnecessary `_amount> 0` check

In `sendReward`, the function sort circuit if `_amount` to be sent is `0`, later however it is checked that this `amount` is `>0` which there is no way it cannot be:

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L1219-L1222

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L1235

```
File: MultiRewardDistributor/MultiRewardDistributor.sol

1219:         // Short circuit if we don't have anything to send
1220:         if (_amount == 0) {
1221:             return _amount;
1222:         }

                          // @audit due to check above there is r
1235:         if (_amount > 0 && _amount <= currentTokenHoldings)
```

Consider removing the `_amount > 0` check since it is always true.

[R-03] `MultiRewardDistributor: IndexUpdate` struct is unnecessary

`IndexUpdate` is used to return the result of `calculateNewIndex`.

However in `calculateNewIndex` the same value for `IndexUpdate.newTimestamp` is always used, `block.timestamp`:

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L949-L953

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L967

```
949:             return
950:                 IndexUpdate({
951:                     newIndex: _currentIndex,
952:                     newTimestamp: blockTimestamp
953:                 });

967:         return IndexUpdate({newIndex: newIndex, newTimestamp
```

Hence, `calculateNewIndex` could simply return `newIndex` (`uin224`). Then in `updateMarketSupplyIndexInternal` and `updateMarketBorrowIndexInternal` the `supply/borrowGlobalTimestamp` can be set directly to `block.timestamp`.

[R-04] Parameter `_mTokenData` for `MultiRewardDistributor::calculateBorrowRewardsForUser` is unnecessary

The struct `MTokenData` has two fields: `mTokenBalance` and `borrowBalanceStored`. However only one of them is used in `calculateBorrowRewardsForUser`.

Therefore it is confusing that the whole struct is passed to the function as that implies that both fields would be used.

Consider just passing `borrowBalanceStored` directly instead of the struct.

Non-critical (6)

[N-01] Wrong grammatical number in `grantGuardiansPause`

`grantGuardiansPause` grants `guardian` pause. Since there is only one `guardian` the name should be singular not plural: `grantGuardianPause`.

[N-02] Erroneous docs

https://github.com/code-423n4/2023-07-moonwell/blob/main/docs/TEMPORALGOVERNOR.md

> Guardian Pause: The guardian has the ability to pause the contract temporarily. When the guardian pauses the contract, all proposal executions are halted. The contract can only be unpaused by a governance proposal after a specified time delay.

This is not true. A governance proposal cannot unpause the `TemporalGovernor`. Unpause can only be performed by `governor` through `togglePause` or by the `permissionlessUnpause` after a delay. The only way governance could unpause is by calling `revokeGuardian` which is final. This would require the `guardian` to execute it though since only `guardian` is allowed to execute commands while the contract is paused.

[N-03] Weird word describing the future

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L788-L792

```
File: MultiRewardDistributor/MultiRewardDistributor.sol

788:         // Must be older than our existing end time AND the
789:         require(
790:             _newEndTime > currentEndTime,
791:             "_newEndTime MUST be > currentEndTime"
792:         );
```

`older` here is confusing. Either say `Must be further in the future` or simply `larger`

[N-04] Forgotten `cToken` references

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L842

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L847

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L881

```
File: MultiRewardDistributor/MultiRewardDistributor.sol

842:         // Calculate change in the cumulative sum of the rew

843:         // Calculate reward accrued: cTokenAmount * accruedI

844:         // Calculate change in the cumulative sum of the rew
```

[N-05] Incorrect comments

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L835-L840

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/MultiRewardDistributor/MultiRewardDistributor.sol#L874-L879

```
File: MultiRewardDistributor/MultiRewardDistributor.sol

835:         // If our user's index isn't set yet, set to the cur
836:         if (
837:             userSupplyIndex == 0 && _globalSupplyIndex >= ir
838:         ) {
839:             userSupplyIndex = initialIndexConstant; //_globa
```

```
840:         }

...

874:         // If our user's index isn't set yet, set to the cur
875:         if (
876:             userBorrowIndex == 0 && _globalBorrowIndex >= ir
877:         ) {
878:             userBorrowIndex = initialIndexConstant; //userBc
879:         }
```

If the index is not yet set it's set to `initialIndexConstant` not the global index as the comment suggests.

[N-06] Comment slash off by one

https://github.com/code-423n4/2023-07-moonwell/blob/main/src/core/Comptroller.sol#L984-L988

```
/** <-- slash only 3 whitespaces in
  * @notice Call out to the reward distributor to update its
  * @param mToken The market to synchronize indexes for
  * @param borrower The borrower to whom rewards are going
  */
```

The slash is only 3 not 4 whitespaces in.

ElliotFriedman (Moonwell) confirmed and commented:

> L-01- correct that guardian can fast track, however this is intended behavior.
> L-02- correct, however this system is only going to be deployed on a single chain.
> L-03- correct, however this doesn't matter as the guardian would not be able to pause or unpause because address 0 is the guardian, so that doesn't work.
> L-04- duplicate finding, this is true, however this is expected behavior.
> L-05- true, but there isn't any negative effect to this happening as an unlisted token in the comptroller cannot be minted or borrowed.

Audit Analysis

For this audit, 12 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The **report highlighted below** by **Sathish9098** received the top score from the judge.

The following wardens also submitted reports: **hals**, **kutugu**, **jaraxxus**, **berlin-101**, **cryptonue**, **catellatech**, **Udsen**, **kodyvim**, **solsaver**, **0xSmartContract**, *and* **K42**.

Summary

List	Head	Details
1	Overview of Moonwell Protocol	overview of the key components and features of Moonwell
2	My Thoughts	My own thoughts about future of the this protocol
3	Audit approach	Process and steps i followed
4	Learnings	Learnings from this protocol
5	Possible Systemic Risks	The possible systemic risks based on my analysis
6	Code Commentary	Suggestions for existing code base
7	Centralization risks	Concerns associated with centralized systems
8	Gas Optimizations	Details about my gas optimizations findings and gas savings
9	Risks as per Analysis	Possible risks
10	Non-functional aspects	General suggestions
11	Time spent on analysis	The Over all time spend for this reports

Overview

`Moonwell Protocol` is a fork of `Compound v2` with features.

Moonwell is an open lending and borrowing protocol built on `Base` , `Moonbeam` , and `Moonriver` .The protocol's intuitive design ensures a seamless user experience, enabling users to easily navigate through various features and perform operations effortlessly.

Important Contracts:

- `ChainlinkCompositeOracle`
    - which aggregates mulitple exchange rates together
- `MultiRewardDistributor`
    - Allow distributing and rewarding users with multiple tokens per MToken. Parts of this system that require special attention are what happens when hooks fail in the Comptroller
- `Comptroller`
    - This contract is based on the Flywheel logic
- `TemporalGovernor`
    - which is the cross chain governance contract. Specific areas of concern include delays, the pause guardian, putting the contract into a state where it cannot be updated

Special features of Moonwell Protocol

- Non-Custodial
- User Experience
- Security
- Transparency
- Onchain Governance

My Thoughts

Moonwell Protocol may changes `Lending and Borrowing` to next level.

Moonwell's approach to lending and borrowing in DeFi can be a catalyst for positive changes in the ecosystem. By setting a standard for user experience, security, transparency, and governance, it may inspire other projects to improve and innovate, leading to a more mature, inclusive, and trustworthy DeFi lending and borrowing landscape in the future.

Decentralization and Governance: Moonwell's onchain governance model can pave the way for more decentralized decision-making processes in other DeFi projects. As protocols adopt similar governance mechanisms, the community's voice and participation may play a more significant role in shaping the future of DeFi platforms.

Trust and Transparency: The emphasis on transparency in Moonwell's onchain operations can foster trust among users. As other protocols adopt similar transparency practices, users may feel more confident in interacting with DeFi platforms, leading to increased trust within the ecosystem.

Non-Custodial Solutions: Moonwell's non-custodial approach to lending and borrowing may become a trend in the DeFi space. More protocols might adopt non-custodial models to allow users to retain control over their assets, aligning with the core principles of DeFi.

Global Reach: Moonwell's focus on accessibility and usability may lead to increased global adoption of DeFi lending and borrowing. As the protocol reaches users in different regions, DeFi's impact could become more widespread and inclusive.

Audit approach

I followed below steps while analyzing and auditing the code base.

1. Read the Audit Readme.md and took the required notes.
2. Moonwell Protocol protocol uses
3. inheritance
4. Tests only covered 80%
5. Multi-Chain
6. ERC-20 Token
7. Timelock function
8. Need to understand moonbeam and temporal governance to understand the governance system
9. Uses Chainlink price oracle
10. This protocol if fork of Compound with MRD
11. Analyzed the over all codebase one iterations very fast

12. Study of documentation to understand each contract purposes, its functionality, how it is connected with other contracts, etc.
13. Then I read old audits and already known findings. Then go through the bot races findings
14. Then setup my testing environment things. Run the tests to checks all test passed. I used foundry to test moonwell protocol. I used forge comments for testing
15. Finally, I started with the auditing the code base in depth way I started understanding line by line code and took the necessary notes to ask some questions to sponsors.

Stages of audit

- `The first stage of the audit`

During the initial stage of the audit for Moonwell Protocol, the primary focus was on analyzing gas usage and quality assurance (QA) aspects. This phase of the audit aimed to ensure the efficiency of gas consumption and verify the robustness of the platform.

Found **14 QA Findings**

- `The second stage of the audit`

In the second stage of the audit for Moonwell Protocol, the focus shifted towards understanding the protocol usage in more detail. This involved identifying and analyzing the important contracts and functions within the system. By examining these key components, the audit aimed to gain a comprehensive understanding of the protocol's functionality and potential risks.

- `The third stage of the audit`

During the third stage of the audit for Moonwell Protocol, the focus was on thoroughly examining and marking any doubtful or vulnerable areas within the protocol. This stage involved conducting comprehensive vulnerability assessments and identifying potential weaknesses in the system. Found `60-70 vulnerable` and `weakness` code parts all marked with `@audit tags` .

- `The fourth stage of the audit`

During the fourth stage of the audit for Moonwell Protocol, a comprehensive analysis and testing of the previously identified doubtful and vulnerable areas were conducted. This stage involved diving deeper into these areas, performing in-depth examinations, and subjecting them to rigorous testing, including fuzzing with various inputs. Finally concluded findings after all research's and tests. Then i reported C4 with proper formats

Found one medium risk findings `admin` **may receive less token than expected because of this check** `_amount == type(uint256).max`

Learnings

The Moonwell Protocol team can draw lessons from other DeFi protocols that have faced similar challenges. Studying security incidents, best practices, and successful governance mechanisms from other projects can provide valuable insights for building a more robust and resilient platform.

By prioritizing security, conducting thorough risk assessments, and actively involving the community in governance decisions, the Moonwell Protocol can proactively address these areas of concern and strengthen its position as a secure and community-driven DeFi lending and borrowing platform

Possible Systemic Risks

- `Smart Contract Vulnerabilities` : The use of inheritance and integration of multiple features can introduce potential smart contract vulnerabilities. Bugs, logic flaws, or improper implementations could lead to exploits and financial losses.
- `Test Coverage Risks` : With only 80% test coverage, the protocol may have undiscovered vulnerabilities. Inadequate security audits could leave critical issues unnoticed, increasing the risk of attacks
- `Governance Challenges` : Understanding Moonbeam and Temporal Governance systems may be complex for some users, potentially leading to governance inefficiencies or suboptimal decision-making.

- `Chainlink Oracle Risks` : Relying on a single oracle provider, like Chainlink, exposes the protocol to potential risks associated with oracle failures, data manipulation, or centralization issues.
- `Protocol Fork Risks` : Forking from other protocols may inherit vulnerabilities present in the original codebase. Failure to address these risks could result in potential attacks.
- `Liquidity and Market Risks` : Insufficient liquidity or market manipulation risks can affect the protocol's stability and overall performance.

Code Commentary

- Use consistant `SPDX-License` for all contracts
- Inconsistance `solidity versions` used
- Ensure that variable and function names follow `consistent naming conventions` for better `readability` and `maintainability` . For example, use camelCase or snake_case consistently throughout the contract
- In the `Status` and `DistributionStatus` structs, consider using an `enum type` for stage to improve readability and avoid potential bugs caused by using numbers directly
- Some functions like `updateMarketSupplyIndexAndDisburseSupplierRewards` and `updateMarketBorrowIndexAndDisburseBorrowerRewards` are similar. You can `consolidate them into a single function` that takes an additional parameter to specify the action (supplier rewards or borrower rewards)
- Instead of importing the `whole` contract code for `IERC20` and `MToken` , use interfaces to define only the `required functions` . This reduces the gas cost and enhances code readability
- Add inline comments to explain complex logic, especially in mathematical calculations, to make the code more understandable
- Add appropriate error messages in require statements to provide more `informative feedback` to users when a transaction fails
- In some functions like `exitMarket` , multiple emit statements are used for the `same event` . `Consolidate` the event emissions to a `single location` to reduce `code duplication` and improve `readability` .

- It's a good practice to add more detailed information in the `event logs`, such as the account addresses involved in certain actions.
- Some functions return error codes `(e.g., uint(Error.NO_ERROR))`, but the exact meaning of these `error codes` is not explicitly defined. Proper error code documentation or the use of error enums would enhance `clarity` and `usability`.
- In the `addressToBytes` function, you can use `bytes20(addr)` directly instead of `casting` it as a `bytes20`. It will not have any impact on gas costs, but it `simplifies the code`
- In the `setTrustedSenders` and `unSetTrustedSenders` functions, avoid encoding and decoding the same data by directly using the addr parameter as the key for the mapping
- `Emit` events for significant `contract state changes`, as it helps in tracking contract activities and provides transparency to external applications
- Consider `refactoring` the code to break down `complex functions` into smaller, more manageable pieces, following the principles of `modularity`
- Instead of using separate bool variables like `guardianPauseAllowed`, you can use an enum to represent different contract states, such as `enum ContractState { Active, Paused, GuardianRevoked }`. This can help make the contract state more explicit and easier to understand

Centralization risks

A single point of failure is not acceptable for this project Centrality risk is high in the project, the role of `onlyOwner` detailed below has very critical and important powers

Project and funds may be compromised by a malicious or stolen private key `onlyOwner` `msg.sender`

```
File: src/core/Governance/TemporalGovernor.sol

266:     function fastTrackProposalExecution(bytes memory VAA) e

274:     function togglePause() external onlyOwner {
```

https://github.com/code-423n4/2023-07-moonwell/blob/4aaa7d6767da3bc42e31c18ea2e75736a4ea53d4/src/core/Governance/TemporalGovernor.sol#L266-L266

Gas Optimizations

1. Using `token.balanceOf(address(this))` every time can potentially reduce the contract's performance. Frequent external calls to other contracts, such as reading the balance of the token contract, can be expensive in terms of gas costs and execution time. This can lead to higher transaction costs and slower contract execution
2. Explicitly `initializing default values` for variables consumes `additional ga` s during contract deployment. If you rely on the EVM's automatic default initialization, you can save gas costs associated with these unnecessary operations
3. `Minimize the number of state variable reads and writes`. Use `local variables` when possible to avoid additional storage operations
4. `Limit` the number of iterations in `loops` to prevent `excessive gas consumption`. Consider using other patterns like `mapping`, where possible, to reduce the need for loops
5. Whenever applicable, consider `batching multiple operations` together to save on gas costs. For example, you can combine multiple token transfers into a single function call
6. Remove any `unused or commented-out code` to save space and simplify contract logic
7. Check if `modifiers` can be `combined` or `reused across multiple functions` to reduce the number of modifier calls. This can save gas by avoiding redundant checks.
8. The `enterMarkets` function currently loops through the list of `mTokens` and `emits an event` for each entry. Consider `batching` the `events` or using other gas-saving techniques to reduce transaction costs
9. In the `addToMarketInternal` function, you can optimize the code by storing `markets[address(mToken)]` in a local variable, rather than repeatedly accessing it within the function

10. Some operations, such as updating `supply and borrow indexes`, are performed in a loop for each market in the `claimReward` function. This could lead to exceeding the block gas limit if the number of markets is substantial
11. Evaluate whether certain loops can be simplified or avoided altogether to reduce gas costs. For instance, consider whether iterating over the allMarkets array in the `_addMarketInternal` function can be optimized
12. `Reuse variables` instead of creating new ones when possible. This can reduce `memory usage` and, in turn, save gas
13. For functions that `don't modify state`, use the `view or pure` modifiers to avoid unnecessary gas costs associated with state changes
14. For `small and simple functions`, consider `inlining` them within other functions to save gas costs associated with function calls
15. Use gas-efficient math libraries, if available, to perform arithmetic operations. Check if there are any `gas-efficient` replacements for the current `math operations`
16. If certain variables, such as `admin, borrowCapGuardian, pauseGuardian,` etc., are not intended to be changed after deployment, mark them as `immutable` to enhance `security` and `gas efficiency`
17. Simplify code logic and eliminate `redundant calculations`
18. In the `executeProposal` function, there are two function calls to `trustedSenders[vm.emitterChainId].contains(vm.emitterAddress)`. Consider storing the result of this call in a variable and using that variable to avoid calling the function `twice`
19. In the `setTrustedSenders` and `unSetTrustedSenders` functions, you can move the `msg.sender == address(this)` check outside the loop to minimize `storage access` in each iteration saves gas
20. In the `constructor`, you can use emit to emit the `TrustedSenderUpdated` event instead of using emit in the loop. This can reduce the gas cost by avoiding multiple event emissions
21. In the `executeProposal` function, the return value `returnData` is not used. You can `remove` it if it's not required.

Risks as per Analysis

- Lack of integer validations in `_setEmissionCap()` function leads unexpected behavior
- if `(_amount == type(uint256).max)` check is wrong . There is no gurantee that `type(uint256).max` and contract balance or same
- `updateMarketSupplyIndexInternal()` , `updateMarketSupplyIndexInternal()` there is no limits checked when updating totalsupply
- `Inline assembly` should be used with caution as it can introduce security risks. Whenever possible, prefer using `built-in Solidity functions` and `libraries` .

`MultiRewardDistributor.sol`

- The contract interacts with external contracts, such as `comptroller, IERC20, and MToken` , through external calls. These external calls are not checked for `success or failure` , and there is no handling for potential exceptions or revert reasons, which can lead to `unexpected results` .
- The contract involves various `mathematical calculations` , such as `index updates` and `reward distributions` . It's crucial to ensure that these operations are handled correctly and do not result in `integer overflows` or `underflows` .
- Some of the functions in the contract, such as `updateMarketSupplyIndexInternal` , `updateMarketBorrowIndexInternal` , `disburseSupplierRewardsInternal` , and `disburseBorrowerRewardsInternal` , perform calculations and iterations that could potentially consume a significant amount of gas. It's essential to ensure that these functions are optimized to prevent gas limitations and high transaction costs.
- The contract uses `uint224` data type for some of the index values `(_globalSupplyIndex, _globalBorrowIndex, etc.)` . If these indices grow `too large` , it could potentially lead to overflow issues and incorrect reward calculations .

- The `sendReward` function performs a token transfer and is marked as `nonReentrant`, but it's essential to ensure that all external contract calls, especially token transfers, are safe against reentrancy attacks
- The contract allows the owner to update `supply` and `borrow emission speeds`. There should be careful consideration of the emission rates and their effects on the overall `token economy` to avoid unintended consequences
- The contract has two functions `(calculateSupplyRewardsForUser and calculateBorrowRewardsForUser)` for calculating rewards for `suppliers and borrowers`, respectively. Ensure that both functions use consistent calculations and share the `same index` values to `prevent discrepancies`
- Consider using `upgradeable contract patterns` to allow for `future updates` without redeploying the entire contract

`Comptroller.sol`

- In the transferAllowed function, there's a comment mentioning the use of local vars to `avoid stack-depth limits` in calculating account liquidity. While it's a good practice to use `local variables` for `complex calculations`, it's important to ensure that the `overall stack depth` is managed appropriately, especially in functions that could be called in `nested scenarios`
- The contract appears to have some administrative functions like `_setPriceOracle, _setCloseFactor, _setCollateralFactor,` etc., which should only be callable by the `admin`. However, the access control checks are not consistently applied in all these functions, potentially allowing unauthorized users to modify critical parameters
- Several functions call external contracts, like the `rewardDistributor`, `oracle`, and token contracts, but they do `not have proper error handling` for potential failures. Lack of error handling can lead to unexpected behavior and vulnerabilities
- The contract has several functions for pausing different actions `(_setMintPaused, _setBorrowPaused, _setTransferPaused, _setSeizePaused)`. However, there is no mechanism for `time-based unpausing or emergency unpause`, which could be essential for the contract's safety

- The `liquidateCalculateSeizeTokens` function, used during liquidation, performs arithmetic calculations using external data (e.g., exchangeRateMantissa). Failing to `handle or validate external data correctly` could lead to vulnerabilities in the liquidation process
- The `claimReward` function, responsible for distributing rewards, has a complex design with nested loops, and it calls an external contract multiple times. High complexity increases the risk of errors and makes the contract harder to audit
- The `getBlockTimestamp` function retrieves the current block timestamp using block.timestamp. It's worth noting that relying solely on block timestamps for time-sensitive operations can be manipulated by miners to some extent
- Certain functions that modify contract parameters (e.g., _setPriceOracle, _setCloseFactor, etc.) do not include `time limitations` for changes. Adding `time restrictions` for critical parameter modifications could prevent `malicious or mistaken changes`

`TemporalGovernor.sol`

- The contract relies on trusted emitters from the Wormhole bridge using `isTrustedSender`. However, relying solely on a `chain ID` and a 32-byte emitter address might not be sufficient to ensure the `authenticity` of the signed messages. Consider using a more `robust signature verification` mechanism to ensure the validity of messages
- The contract doesn't have any explicit `reentrancy protection`. Ensure that critical state changes are performed before any external calls to prevent potential `reentrancy attacks`
- The contract uses `uint248` for `lastPauseTime`, which might be susceptible to integer overflow if the pause timestamp becomes larger than the `maximum value`
- The fastTrackProposalExecution function allows the owner to bypass the usual delay for proposal execution. This can be risky as it doesn't have any checks on the validity of the proposal. It should be handled with caution and only used in specific emergency scenarios
- The function `allTrustedSender`s iterates through the entire trustedSenders set to return the list of trusted senders for a chain. This operation may become

inefficient as the size of the set grows. Consider alternative data structures or optimizations if needed

`ChainlinkCompositeOracle.sol`

- The contract relies on `Chainlink oracles` to provide accurate price data. If a Chainlink oracle is `compromised` or `provides incorrect data`, the ChainlinkCompositeOracle contract could also be compromised
- The contract does not have any mechanisms in place to `prevent front-running` or `other forms of market manipulation`. This could allow malicious actors to exploit the contract and profit at the expense of other users
- The contract relies on the `decimals` value returned by Chainlink oracles. If the decimals value is misaligned with the `contract's expectations`, it could result in `incorrect price calculations` and pose a vulnerability

Non-functional aspects

- Aim for high test coverage to validate the contract's behavior and catch potential bugs or vulnerabilities
- The protocol execution flow is not explained in efficient way.
- Its best to explain over all protocol with architecture is easy to understandings
- Consider designing the contract to be upgradable or allow for versioning. This can help address issues, introduce new features, or adapt to evolving requirements without disrupting the entire system

Time spent on analysis

`15 Hours`

ElliotFriedman (Moonwell) acknowledged

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top