SHERLOCK

# Sherlock Security Review For Morph L2

| | |
|---|---|
| Public contest prepared for: | **Morph L2** |
| Lead Security Expert: | **mstpr-brainbot** |
| Date Audited: | **September 2 - September 23, 2024** |

## Introduction

Morph is an optimistic zkevm layer2 with decentralized sequencers. We'll focus on the security of fund in smart contracts like bridge and staking. In terms of marketing, Morph is the Consumer Blockchain for Everyday Life.

## Scope

Repository: morph-l2/morph

Branch: main

Audited Commit: 22ca805e2d09c9d0bddb3e8a52ddd7d3435ce769

Final Commit: c56fb938a8a3da81d1e8979b2c49a9d7b3acc7ec

---

For the detailed scope, see the contest details.

## Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

## Issues found

| Medium | High |
|---|---|
| 11 | 3 |

## Issues not fixed or acknowledged

| Medium | High |
|---|---|
| 0 | 0 |

## Security experts who found valid issues

PapaPitufo
sammy
mstpr-brainbot
0xRajkumar
0xStalin
n4nika
HaxSecurity
PASCAL
underdog
p0wd3r
ulas
zraxx
DeltaXV

## Issue H-1: Incorrect implementation of the onDropMessage function in the L1ReverseCustomGateway contract

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/66

### Found by

0xRajkumar, mstpr-brainbot, sammy

### Summary and Vulnerability Detail

In the L1ReverseCustomGateway, tokens are burned during deposit and minted during the finalizeWithdrawERC20 process.

For failed deposits, we have the onDropMessage function to allow the user to retrieve their tokens.

```
function onDropMessage(bytes calldata _message) external payable virtual
↪ onlyInDropContext nonReentrant {
    // _message should start with 0x8431f5c1  =>
↪   finalizeDepositERC20(address,address,address,address,uint256,bytes)
    require(bytes4(_message[0:4]) == IL2ERC20Gateway.finalizeDepositERC20.selector,
↪   "invalid selector");

    // decode (token, receiver, amount)
    (address _token, , address _receiver, , uint256 _amount, ) = abi.decode(
        _message[4:],
        (address, address, address, address, uint256, bytes)
    );

    // do dome check for each custom gateway
    _beforeDropMessage(_token, _receiver, _amount);

    IERC20Upgradeable(_token).safeTransfer(_receiver, _amount);

    emit RefundERC20(_token, _receiver, _amount);
}
```

It will not work now because instead of using safeTransfer we should mint the tokens to _receiver.

## Impact

The impact is high because the dropMessage functionality will never work, preventing the user from recovering their tokens.

## Tool used

Manual Review

## Recommendation

My recommendation is to use `IMorphERC20Upgradeable(_token).mint(_receiver,_amount)` instead of transferring the tokens.

## References

https://github.com/sherlock-audit/2024-08-morphl2/blob/main/morph/contracts/contracts/l1/gateways/L1ReverseCustomGateway.sol

https://github.com/sherlock-audit/2024-08-morphl2/blob/main/morph/contracts/contracts/l1/gateways/L1ERC20Gateway.sol#L74-L90

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/morph-l2/morph/pull/562

## Issue H-2: Attacker can freeze chain and steal challenge deposits using fake `prevStateRoot`

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/84

### Found by

PapaPitufo

### Summary

Because the `prevStateRoot` is not validated until a batch is finalized, a committed batch with a malicious `prevStateRoot` can be used to both (a) win challenges against honest challengers and (b) halt the chain since it will be approved but be unable to be finalized.

### Root Cause

In `Rollup.sol`, if a malicious batch is proposed, the assumption is that the sequencer who proposed it will lose the challenge, get slashed, and the chain will be reset. These economic incentives prevent the chain from being regularly halted.

This is based on the assumption that a sequencer can only win challenges if the batch they proposed is valid.

However, the check that `prevStateRoot` is actually the `postStateRoot` of the previous batch only happens in `finalizeBatch()`. This check is sufficient to prevent batches with fake `prevStateRoots` from being finalized, but it does not stop these batches from being committed.

This allows a malicious sequencer to propose any batch that performs a valid state transaction on a fake `prevStateRoot`.

In most cases, a challenger will attack this invalid batch. However, it is possible for the sequencer to provide a valid proof of this state transition to steal the honest challenger's deposit and win the challenge.

In the case that this happens, or that no challenge is performed, the committed batch will not be able to finalized due to the following check:

```
require(
    finalizedStateRoots[_batchIndex - 1] ==
↪   BatchHeaderCodecV0.getPrevStateHash(memPtr),
    "incorrect previous state root"
);
```

This will freeze the chain and not allow any new batches to be finalized, since batches are committed sequentially and must be finalized sequentially.

## Internal Preconditions

None

## External Preconditions

None

## Attack Path

1. Attacker proposes a batch that contains a valid state transition from a fake `prevStateRoot`.
2. If an honest challenger challenges the batch, the attacker provides a valid proof of the state transition to win the challenge and steal the challenger's deposit.
3. Whether or not the above happens, the chain is now halted, as the attacker's batch cannot be finalized, and no other batches can be finalized without it being finalized first.
4. The attacker will not be slashed, due to the fact that they won the challenge.

## Impact

- An honest challenge will lose their deposit when a dishonest sequencer beats them in a challenge.
- No new batches will be able to be finalized, so the chain will halt and have to be manually rolled back by the admins.

## PoC

N/A

## Mitigation

Check in `commitBatch()` that `prevStateRoot` is equal to the `parentBatchHeader.postStateRoot`.

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/morph-l2/morph/pull/616

## Issue H-3: Sequencer will be underpaid because of incorrect `commitScalar`

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/90

The protocol has acknowledged this issue.

### Found by

PapaPitufo

### Summary

`GasPriceOracle.sol` underprices the cost of calling `commitBatch()`, so the `l1DataCost` paid by users will be substantially less than the true cost to the sequencer. This is made worse when transactions are heavily weighted towards L1 cost, in which case the sequencer can be responsible for payments 100X as large as the revenue collected.

### Root Cause

When new L2 transactions are submitted, the transaction cost is calculated as the L2 gas price plus an `l1DataFee`, which is intended to cover the cost of posting the data to L1.

The `l1DataFee` is actually calculated in go-ethereum rollup/fees/rollup_fee.go#L153, but is equivalent to this calculation in `GasPriceOracle.sol` as:

```
function getL1FeeCurie(bytes memory _data) internal view returns (uint256) {
    // We have bounded the value of `commitScalar` and `blobScalar`, the whole
 ↪  expression won't overflow.
    return (commitScalar * l1BaseFee + blobScalar * _data.length * l1BlobBaseFee) /
 ↪  PRECISION;
}
```

We can summarize as:

- We pay 1 gas per byte that goes into a blob (because GAS_PER_BLOB == 2**17), so we can calculate the cost of the blob as `blobScalar*_data.length*l1BlobBaseFee`.
- We have to call `commitBatch()` as well, so we take the gas cost of that call and multiply by the `l1BaseFee`.

The config specifies that the `blobScalar=0.4e9` and the `commitScalar=230e9`.

This `commitScalar` is underpriced by a margin of 100X, as it should be `230_000e9`, not `230e9`.

The result is that transactions that are largely weighted towards the L1 fee will cost the sequencer way more to commit to than the user paid.

In an extreme, we can imagine an attack where users performed the most L1 data intensive transactions possible, which consist of filling a full batch with a single transaction that uses 128kb of calldata.

Let's look at the relative costs for the attacker and the sequencer of such a transaction:

- The attacker's fee is calculated as `intrinsicgas+calldata+l1DataFee`.
- Since blobs are 128kb, we need `1024*128=131,072` bytes of calldata to fill the blob.
- Assuming we need half non-zero to avoid compression, we can calculate the calldata cost as: `1024*128*(4+16/2)=1,310,720`.
- Therefore, the total gas used by the attacker is `21_000+1,310,720=1,331,720`.
- The l1DataFee is calculated as above. If we assume an l1BaseFee of 30 gwei, this gives a cost of `52,428+(230*30)=59,328gwei`.

- Assuming an L2 gas price of 0.001 gwei (estimate from L1 message queue), our total cost is `(1,331,720*0.001)+59,328=60,659gwei=0.13`.

On the other hands, let's look at the sequencer's cost:

- The blob cost is the same as the attacker's l1DataFee, as that is calculated correctly at `52,428gwei`.
- The transaction cost, based on the Foundry estimate, is 230,000 gas. At 30 gwei per gas, that equals `230_000*30=6,900,000gwei`.

- The total cost to the sequencer is `6,900,000+52,428=6,952,428gwei=15.98`.

This mismatch requires sequencers to spend more than 100X to progress the chain, when only X was collected in sequencer fees from the user.

## Internal Preconditions

None

## External Preconditions

None

## Attack Path

Note that this mispricing will cause the chain to slowly bleed revenue, regardless of any ”attack”. However, the most accute version of this attack would be as follows:

1. User send a transfer of 0 value on L2 with 128kb of calldata.
2. This costs them approximately $0.13 in L2 fees, but will require the sequencer to spent $15.98 in L1 gas to commit the batch.

3. The user can repeat this attack indefinitely, causing the sequencer to spend more money than it receives.

## Impact

Sequencers can be forced to spend 100X more than the fee revenue received in order to progress the chain, making the chain uneconomical.

## PoC

N/A

## Mitigation

`commitBatch()` should be set to `230_000e9`.

Note that it appears that this value was pulled from Scroll's current on chain implementation. This issue has been reported directly to Scroll as well, and has been deemed valid and awarded with a bounty via their Immunefi bounty program.

## Issue M-1: `Rollup.sol` cannot split batches across blobs, allowing inexpensive block stuffing

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/91

The protocol has acknowledged this issue.

### Found by

PapaPitufo

### Summary

The maximum amount of calldata that can be passed to an L2 transaction is too large to fit in a single blob. Because `Rollup.sol` does not allow splitting batches across blobs, L2 blocks are capped by their calldata size. This is not properly accounted for in the L2 gas prices, and leads to an attack where blocks can be inexpensively stuffed, blocking all L2 transactions on the chain.

### Root Cause

Batches are expected to commit to many blocks. We can expect up to 100 blocks per chunk (`MaxBlocksPerChunk` in node/types/chunk.go), and 45 chunks per batch (`maxChunks` in Gov.sol). This means that a batch can commit to 4500 blocks.

However, `Rollup.sol` has the surprising quirk that a full batch must fit into a single blob. For that reason, the batch is not just limited based on blocks, but also by calldata. We can see this logic in miner/pipeline.go#L260-266, where the block size is being tracked, and we skip all transactions that push it over the limit for the blob.

In the event that a user submits a single transaction with enough calldata to fill a whole blob, this transaction will end the block, and the entire batch will consist of the single block with one transaction.

This has two important implications:

First, the gas cost of stuffing an entire block is loosely the price of sending 128kb of calldata. For a transaction with no execution or value, we can calculate the L2 gas cost as `intrinsicgas+calldata+l1DataFee`.

If we assume an l1BaseFee of 30 gwei, an l1BlobBaseFee of 1, and the scalar and l2BaseFee values from the config file, we get:

- `intrinsicgas=21_000gas=21gwei`

    - `calldata=1024*128*(4+16)/2=1,310,720gas=1,310gwei` (assumes half non-zero bytes to avoid compression)
    - `l1DataFee=(1024*128*1*0.4)+(230*30)=59,328gwei`
- `totalcost=21+1,310+59,328=60,659gwei=`0.14

Second, and more importantly, block stuffing is usually protected by EIP1559 style gas pricing, where the base fee increases dramatically if sequential blocks are full. However, this block stuffing attack has the strange quirk that only 1.3mm gas (out of 30mm gas limit) will be used, which will actually lower the base fee over time.

## Internal Preconditions

None

## External Preconditions

None

## Attack Path

1. Submit a large number of transactions on L2 that use 128kb of calldata.
2. Each time one is picked up (which should be each block), it costs only 1.3mm gas, but seals the block with no other L2 transactions.

## Impact

L2 blocks will be stuffed and most activity on L2 will be blocked. This can cause major and unpredictable issues, such as oracles getting out of date, liquidations not occurring on time, and users being unable to take other important actions.

## PoC

N/A

## Mitigation

The architecture should be changed so that batches can be split across multiple blobs if needed.

## Issue M-2: Possible wrong accounting in L1Staking.sol

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/104

The protocol has acknowledged this issue.

### Found by

0xRajkumar, PASCAL, PapaPitufo, n4nika

### Summary

Possible wrong accounting in L1Staking.sol during some slashing occasions.

### Vulnerability Detail

Stakers are permitted to commit batches in the rollup contract and these batches can be challenged by challengers, if the challenge is successful the challenger gets part of the stakers ETH and staker is removed, the owner also takes the rest; if it wasn't succesful the prover takes all the challengeDeposit from the challenger. According to the readMe https://github.com/sherlock-audit/2024-08-morphl2-Pascal4me#q-are-there-any-limitations-on-values-set-by-admins-or-other-roles-in-the-codebase-including-restrictions-on-array-lengths `proofWindow` which is the time a challenged batch has to go without being proven for it to be successfully challenged can be set to as high as 604800 seconds which is 7 days and `withdrawalLockBlocks` default value is also 7 days(No relation between the two was done and owner can change `proofWindow` value in rollup contract). For this vulnerability we'll assume `proofWindow` is set to 7 days. The issue stems from a staker being able to commit a wrong batch and still being able to withdraw from the staking contract without that batch being finalized or proven. Let's site this example with `proofWindow` being set to 7 days

- Alice a staker commits a wrong batch and immediately goes ahead to withdraw her ETH from the staking contract, her 7 days period countdown start
- Bob a challenger sees that wrong batch and challenges it and it since it's a wrong batch theres no proof for it, then 7 days couuntdown starts. But remember the withdraw function 7 days started counting first so when it elapses Alice quickly goes ahead to withdraw her ETH, then when Bob calls `proveState()`, Alice is supposedly slashed but it's useless as she's already left the system. Then the contract is updated as though Alice's ETH is still in the contract

```
 uint256reward=(valueSum*rewardPercentage)/100;slashRemaining+=valueSum-reward;
_transfer(rollupContract,reward);
```

So a current staker's ETH is the one being sent to the rollup contract and being assigned to the owner via `slashRemaining`. So there's less ETH than the contract is accounting for which already an issue. This will be detrimental when all the ETH is being withdrawn by stakers and owner the last person transaction will revert becuase that ETH is not in the contract.

## Impact

Incorrect contract accounting

## Code Snippet

https://github.com/sherlock-audit/2024-08-morphl2/blob/main/morph/contracts/contracts/l1/staking/L1Staking.sol#L197-L201 https://github.com/sherlock-audit/2024-08-morphl2/blob/main/morph/contracts/contracts/l1/rollup/Rollup.sol#L484-L487 https://github.com/sherlock-audit/2024-08-morphl2/blob/main/morph/contracts/contracts/l1/rollup/Rollup.sol#L697-L707 https://github.com/sherlock-audit/2024-08-morphl2/blob/main/morph/contracts/contracts/l1/staking/L1Staking.sol#L307-L311 https://github.com/sherlock-audit/2024-08-morphl2/blob/main/morph/contracts/contracts/l1/staking/L1Staking.sol#L217-L237

## Tool used

Manual Review

## Recommendation

Ensure stakers can't withdraw if they have a batch that is unfinalized or unproven and always ensure that withdraw time block is > `proofWindow`

## Issue M-3: The 255th staker in `L1Staking.sol` can avoid getting slashed and inadvertently cause fund loss to stakers

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/142

### Found by

0xRajkumar, p0wd3r, sammy, zraxx

### Summary

The 255th staker in `L1Staking.sol` can avoid getting slashed and inadvertently cause fund loss to stakers

### Vulnerability Detail

The `L1Staking.sol` contract supports upto 255 stakers :

```
/// @notice all stakers (0-254)
address[255] public stakerSet;

/// @notice all stakers indexes (1-255). '0' means not exist. stakerIndexes[1]
↪ releated to stakerSet[0]
mapping(address stakerAddr => uint8 index) public stakerIndexes;
```

Everytime a staker registers in `L1Staking.sol` they are added to the `stakerSet` and their index is stored in `stakerIndexes` as `index+1`

These stakers, while active, can commit batches in `Rollup.sol` using `commitBatch()` and the `batchDataStore[]` mapping is updated as follows :

```
batchDataStore[_batchIndex] = BatchData(
    block.timestamp,
    block.timestamp + finalizationPeriodSeconds,
    _loadL2BlockNumber(batchDataInput.chunks[_chunksLength - 1]),
    // Before BLS is implemented, the accuracy of the sequencer set uploaded by
↪   rollup cannot be guaranteed.
    // Therefore, if the batch is successfully challenged, only the submitter will
↪   be punished.
    IL1Staking(l1StakingContract).getStakerBitmap(_msgSender()) // =>
↪   batchSignature.signedSequencersBitmap
);
```

On a successful challenge, the `_challengerWin()` function is called, and here the `sequencersBitmap` is the one that was stored in `batchDataStore[]`

```
function _challengerWin(uint256 batchIndex, uint256 sequencersBitmap, string memory
 ↪  _type) internal {
    revertReqIndex = batchIndex;
    address challenger = challenges[batchIndex].challenger;
    uint256 reward = IL1Staking(l1StakingContract).slash(sequencersBitmap);
    batchChallengeReward[challenges[batchIndex].challenger] +=
 ↪  (challenges[batchIndex].challengeDeposit + reward);
    emit ChallengeRes(batchIndex, challenger, _type);
}
```

This function calls the `slash()` function in `L1Staking.sol`:

```
function slash(uint256 sequencersBitmap) external onlyRollupContract nonReentrant
 ↪  returns (uint256) {
    address[] memory sequencers = getStakersFromBitmap(sequencersBitmap);

    uint256 valueSum;
    for (uint256 i = 0; i < sequencers.length; i++) {
        if (withdrawals[sequencers[i]] > 0) {
            delete withdrawals[sequencers[i]];
            valueSum += stakingValue;
        } else if (!isStakerInDeleteList(sequencers[i])) {
            // If it is the first time to be slashed
            valueSum += stakingValue;
            _removeStaker(sequencers[i]);
            // remove from whitelist
            delete whitelist[sequencers[i]];
            removedList[sequencers[i]] = true;
        }
    }

    uint256 reward = (valueSum * rewardPercentage) / 100;
    slashRemaining += valueSum - reward;
    _transfer(rollupContract, reward);

    emit Slashed(sequencers);
    emit StakersRemoved(sequencers);

    // send message to remove stakers on l2
    _msgRemoveStakers(sequencers);

    return reward;
}
```

The function converts the `sequencersBitmap` into an array by calling :

```
function getStakersFromBitmap(uint256 bitmap) public view returns (address[] memory
 ↪  stakerAddrs) {
    // skip first bit
    uint256 _bitmap = bitmap >> 1;
    uint256 stakersLength = 0;
    while (_bitmap > 0) {
        stakersLength = stakersLength + 1;
        _bitmap = _bitmap & (_bitmap - 1);
    }

    stakerAddrs = new address[](stakersLength);
    uint256 index = 0;
    for (uint8 i = 1; i < 255; i++) {
        if ((bitmap & (1 << i)) > 0) {
            stakerAddrs[index] = stakerSet[i - 1];
            index = index + 1;
            if (index >= stakersLength) {
                break;
            }
        }
    }
}
```

Since `bitmap` will only contain 1 staker's bit, the `stakersLength` here will be 1. The loop then checks every single bit of the bitmap to see if it's active. Notice, however, that `i` only goes up to 254, and 255 is skipped. This means that for the 255th staker having index of `254`, the array will contain `address(0)`.

This means that in `slash()`, this code will be execued :

```
else if (!isStakerInDeleteList(sequencers[i])) {
                // If it is the first time to be slashed
                valueSum += stakingValue;
                _removeStaker(sequencers[i]);
                // remove from whitelist
                delete whitelist[sequencers[i]];
                removedList[sequencers[i]] = true;
            }
```

`_removeStaker()` is called with `addr=address(0)`:

```
function _removeStaker(address addr) internal {
    require(deleteableHeight[addr] == 0, "already in deleteList");
    deleteList.push(addr);
    deleteableHeight[addr] = block.number + withdrawalLockBlocks;
}
```

This means that the staker avoids being removed and the intended state changes are made to `address(0)` instead. The staker can continue committing invalid batches to the Rollup and not get slashed. Additionally, the `stakingValue` is still rewarded to the challenger, while the staker isn't actually removed from the protocol. Over time, the ETH of `L1Staking.sol` will run out and it won't be possible for stakers to withdraw or for them to get slashed.

## Impact

Critical - loss of funds and breaks protocol functionality

## Coded POC

```
    function test_poc_255() external{
         address[] memory add = new address[](255);

         for(uint256 i = 0 ; i < 255 ; i++)
         {
            add[i] = address(bytes20(bytes32(keccak256(abi.encodePacked(1500 +
↪   i)))));
         }

        hevm.prank(multisig);
        l1Staking.updateWhitelist(add, new address[](0));

        // register all the 255 stakers
        for(uint256 i = 0 ; i < 255 ; i++)
         {
        Types.StakerInfo memory info;
        info.tmKey = bytes32(i+1);
        bytes memory blsKey = new bytes(256);
        blsKey[31] = bytes1(uint8(i));
        info.blsKey = blsKey;
        assert(info.blsKey.length == 256);
        hevm.deal(add[i], 5 * STAKING_VALUE);
        hevm.prank(add[i]);
        l1Staking.register{value: STAKING_VALUE}(info.tmKey, info.blsKey);
         }

        assertEq(add.length, 255);
        address[] memory arr = new address[](1);
        arr[0] = add[254];
         uint256 _bitmap = l1Staking.getStakersBitmap(arr); // this bitmap will
↪   contain the 255th staker only
        address[] memory stakers = l1Staking.getStakersFromBitmap(_bitmap);

        // as you can see the array is {address(0)}
        assertEq(stakers[0], address(0));
        // simulate the challenger win flow
        hevm.prank(l1Staking.rollupContract());
        uint256 balanceBefore = address(l1Staking).balance;
        uint256 reward = l1Staking.slash(_bitmap);
        uint256 balanceAfter = address(l1Staking).balance;

        // the contract loses "reward" amount of ETH
        assertEq(balanceBefore, balanceAfter + reward);

        // the 255th staker still remains an active staker
        assert(l1Staking.isActiveStaker(arr[0]) == true);
}
```

To run the test, copy the above in `L1Staking.t.sol` and run `forgetest--match-test"test_poc_255"`

## Code Snippet

## Tool used

Manual Review

## Recommendation

Make the following change :

```
-       for (uint8 i = 1; i < 255; i++) {
+       for (uint8 i = 1; i <= 255; i++) {
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/morph-l2/morph/pull/562

## Issue M-4: Batches committed during an on going challenge can avoid being challenged

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/143

### Found by

PapaPitufo, sammy NOTE : This finding is not the same as Known issue # 10, that is about an invalid batch getting finalized due to `challengeState()` reverting (OOG error)

### Summary

Batches committed during an ongoing challenge can avoid being challenged and pre-maturely finalize if the defender wins

### Vulnerability Detail

After a batch is committed, there is a finalization window, in which challengers can challenge the batch's validity, after the period has elapsed, the batch can be finalized.

A batch can be challenged using `challengeState()` :

```
function challengeState(uint64 batchIndex) external payable onlyChallenger
↪ nonReqRevert whenNotPaused {
    require(!inChallenge, "already in challenge");
    require(lastFinalizedBatchIndex < batchIndex, "batch already finalized");
    require(committedBatches[batchIndex] != 0, "batch not exist");
    require(challenges[batchIndex].challenger == address(0), "batch already
↪   challenged");
    // check challenge window
    require(batchInsideChallengeWindow(batchIndex), "cannot challenge batch outside
↪   the challenge window");
    // check challenge amount
    require(msg.value >= IL1Staking(l1StakingContract).challengeDeposit(),
↪   "insufficient value");

    batchChallenged = batchIndex;
    challenges[batchIndex] = BatchChallenge(batchIndex, _msgSender(), msg.value,
↪   block.timestamp, false, false);
    emit ChallengeState(batchIndex, _msgSender(), msg.value);

    for (uint256 i = lastFinalizedBatchIndex + 1; i <= lastCommittedBatchIndex;
↪   i++) {
        if (i != batchIndex) {
            batchDataStore[i].finalizeTimestamp += proofWindow;
        }
    }

    inChallenge = true;
}
```

As you can see, the function loops through all the unfinalized batches, except the batch being challenged and adds a `proofWindow` to their finalization timestamp. This is to compensate for the amount of time these batches cannot be challenged, which is the duration of the current challenge, i.e, `proofWindow` (only one batch can be challenged at a time).

However,  allows batches to get committed even when a challenge is going on and does not compensate for time spent inside the challenge :

```
batchDataStore[_batchIndex] = BatchData(
    block.timestamp,
    block.timestamp + finalizationPeriodSeconds,
    _loadL2BlockNumber(batchDataInput.chunks[_chunksLength - 1]),
    // Before BLS is implemented, the accuracy of the sequencer set uploaded by
 ↪  rollup cannot be guaranteed.
    // Therefore, if the batch is successfully challenged, only the submitter will
 ↪  be punished.
    IL1Staking(l1StakingContract).getStakerBitmap(_msgSender()) // =>
 ↪  batchSignature.signedSequencersBitmap
);
```

As you can see, the new batch can be finalized after `finalizationPeriodSeconds`.

Currently the value of `finalizationPeriodSeconds` is `86400`, i.e, `1Day` and `proofWindow` is `172800`, i.e, `2Days` Source. This means that a batch committed just after the start of a challenge will be ready to be finalized in just 1 day, before the ongoing challenge even ends.

Now, consider the following scenario :

- A batch is finalized
- A challenger challenges this batch, challenge will end 2 days from the start
- A staker commits a new batch
- After 1 day, this new batch is ready to be finalized, but can't be finalized yet as the parent batch (the challenged batch) needs to be finalized first
- After 1.5 days, the original batch finishes its challenge, the defender wins(by providing a valid ZK proof), and the batch is ready to be finalized
- Right after the original batch is finalized, the new batch is finalized

This leaves no time for a challenger to challenge the new batch, and this can lead to invalid batches getting committed, even if the batch committer (sequencer) doesn't act maliciously.

Since `finalizeBatch()` is permissionless and only checks whether a batch is in the finalization window, anyone can batch the two `finalizeBatch()` calls which finalize both the original batch and the invalid batch, right after the challenge ends (by back running `proveState()`), leaving no time for a challenger to call `challengeState()`

If a sequencer is malicious, they can easily exploit this to commit invalid batches

## Impact

Critical - Can brick the entire Morph L2 protocol

## Code Snippet

## Tool used

Manual Review

## Recommendation

You can make the following change, which correctly compensates for the lost finalization time:

```
            batchDataStore[_batchIndex] = BatchData(
                block.timestamp,
-                block.timestamp + finalizationPeriodSeconds,
+                 block.timestamp + finalizationPeriodSeconds + (inChallenge ?
↪   proofWindow - (block.timestamp - challenges[batchChallenged].startTime) : 0),
                _loadL2BlockNumber(batchDataInput.chunks[_chunksLength - 1]),
                // Before BLS is implemented, the accuracy of the sequencer set
↪   uploaded by rollup cannot be guaranteed.
                // Therefore, if the batch is successfully challenged, only the
↪   submitter will be punished.
                IL1Staking(l1StakingContract).getStakerBitmap(_msgSender()) // =>
↪   batchSignature.signedSequencersBitmap
            );
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/morph-l2/morph/pull/577

## Issue M-5: Malicious challenger can brick `finalizeTimestamp` of unfinalized batches

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/145

The protocol has acknowledged this issue.

### Found by

DeltaXV, PASCAL, n4nika, ulas

### Summary

A malicious challenger can brick the delay for the finalization of batches. This is possible due to the uncapped extension of the finalization period for all unfinalized batches within the `challengeState` function.

### Vulnerability Detail

The `challengeState` function extends the `finalizeTimestamp` by doing an addition with the `proofWindow` variable (2 days) for each index (all unfinalized batches) except the challenged one, with not a single check or safeguard:

```
for (uint256 i = lastFinalizedBatchIndex + 1; i <= lastCommittedBatchIndex; i++) {
    if (i != batchIndex) {
        batchDataStore[i].finalizeTimestamp += proofWindow;
    }
}
```

The permissionless `proveState` function allows anyone including challengers to provide a ZK-proof and immediate resolution of a challenge, setting inChallenge to false:

```
// Mark challenge as finished
challenges[_batchIndex].finished = true;
inChallenge = false;
```

Attack details:

- An attack window of 15 minutes (could be longer)
  - 15 minutes = ~ 75 Ethereum blocks (12 seconds per block)
  - Challenge-prove cycle (due to `inChallenge` flag) => 2 blocks per cycle => 37 batches

  - Each cycle extends the finalization period for all other batches by `proofWindow` = 2 days (37 cycles * 2 days per cycle) means that each batch would inccur an additional extension of ~74 days!
- Attackers capital: 37 ETH => while risking only 1 ETH!
  - Or maybe not, because the malicious challenger could get his deposit back whenever admin pauses the contract see Rollup::L444
  - `claimReward` function doesn't have a `whenNotPaused` modifier meaning that the owner has no ability to freeze attackers funds see Rollup::L543

**This combination allows an attacker to:**

1. Pre-generate ZK proofs for multiple batches (32 unfinalized batch)
2. Call `challengeState` function - Initiate a challenge on a batch
3. Call `proveState` function - Immediately prove the batch in the next block

- Repeat steps 2-3 for multiple batches (32 times)

Result: Each cycle extends the finalization period for all other batches by proofWindow (2 days), `finalizeTimestamp` accumulate a significant delay due to repetitive extension.

## Impact

- Massive delay for unfinalized batches (uncapped delay) - Huge impact on L2 - Short-term: loss of time and money

## Code Snippet

https://github.com/sherlock-audit/2024-08-morphl2/blob/98e0ec4c5bbd0b28f3d3a9e9159d1184bc45b38d/morph/contracts/contracts/l1/rollup/Rollup.sol#L383

## Tool used

Manual Review

## Recommendation

Fix the current design by implementing some of these mitigation (mainly safguards):

- Implement a cooldown period between challenges for the same challenger address
- Cap the maximum extension of the finalization period
- Check which unfinalized batches requires a extension within the loop

- Break this attack incentive ? - Implement withdrawal lock period for withrawal request to monitor any suspicious activity - Add a `whenNotPaused` modifier on the `claimReward` function, incase of exploit could freeze funds
- Ultimately restrict the ”prover” actor (mitigation from sponsor discussed in private thread).

## Issue M-6: A batch can be unintentionally be challenged during L1 reorg leading to loss of funds

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/146

### Found by

HaxSecurity, PASCAL, PapaPitufo, sammy

### Summary

A batch can be unintentionally be challenged during L1 reorg leading to loss of funds

### Root Cause

The incorrect batch can be challenged during L1 reorg leading to loss of funds. Firstly the README states:

> But if there is any issue about Ethereum L1 re-org leading to financial loss, that issue is valid.

In the `challengeBatch` function, the batch that is challenged is referenced by the `batchIndex`.

Rollup.sol#L366-L388

```
/// @dev challengeState challenges a batch by submitting a deposit.
function challengeState(uint64 batchIndex) external payable onlyChallenger
↪  nonReqRevert whenNotPaused {
    require(!inChallenge, "already in challenge");
    require(lastFinalizedBatchIndex < batchIndex, "batch already finalized");
    require(committedBatches[batchIndex] != 0, "batch not exist");
    require(challenges[batchIndex].challenger == address(0), "batch already
↪   challenged");
    // check challenge window
    require(batchInsideChallengeWindow(batchIndex), "cannot challenge batch outside
↪   the challenge window");
    // check challenge amount
    require(msg.value >= IL1Staking(l1StakingContract).challengeDeposit(),
↪   "insufficient value");

    batchChallenged = batchIndex;
    challenges[batchIndex] = BatchChallenge(batchIndex, _msgSender(), msg.value,
↪   block.timestamp, false, false);
```

```
    emit ChallengeState(batchIndex, _msgSender(), msg.value);

    for (uint256 i = lastFinalizedBatchIndex + 1; i <= lastCommittedBatchIndex;
 ↪  i++) {
        if (i != batchIndex) {
            batchDataStore[i].finalizeTimestamp += proofWindow;
        }
    }

    inChallenge = true;
}
```

However, this poses a problem because a reorg can cause the batch present in the `committedBatches[batchIndex]` to change and the challenger unintentionally challenging the incorrect batch, losing the challenge and their ETH.

For instance, consider the scenario where the sequencers upload two different batches at the same `batchIndex`, a correct batch and an incorrect batch. The initial transaction ordering is:

1. Transaction to upload a incorrect batch at `batchIndex=x`
2. Transaction to upload a correct batch (it will revert, as it is already occupied) at `batchIndex=x`
3. Challenger calls `challengeState` at `batchIndex=x`.

An L1 reorg occurs, resulting in the new transaction ordering

1. Transaction to upload a correct batch (it will revert, as it is already occupied) at `batchIndex=x`
2. Transaction to upload a incorrect batch at `batchIndex=x`
3. Challenger calls `challengeState` at `batchIndex=x`.

Due to the L1 reorg, the challenger will now be challenging a correct batch and will proceed to lose their challenge stake as it can be proven by the sequencer.

This issue is really similar to the Optimism reorg finding: https://github.com/sherlock-audit/2024-02-optimism-2024-judging/issues/201, where the incorrect state root can also be challenged leading to loss of bonds.

## Internal pre-conditions

1. L1 reorg

## External pre-conditions

n/a

## Attack Path

n/a

## Impact

A batch can be unintentionally challenged leading to loss of funds for the challenger

## PoC

*No response*

## Mitigation

In the `challengeState` function, allow loading the `batchHeader` and verifying that the `committedBatches[_batchIndex]` is equal to the `_batchHash` as is done in the other functions such as `revertBatch`

```
   /// @dev challengeState challenges a batch by submitting a deposit.
-    function challengeState(uint64 batchIndex) external payable onlyChallenger
↪  nonReqRevert whenNotPaused {
+    function challengeState(bytes calldata _batchHeader) external payable
↪  onlyChallenger nonReqRevert whenNotPaused {
         require(!inChallenge, "already in challenge");

+        (uint256 memPtr, bytes32 _batchHash) = _loadBatchHeader(_batchHeader);
+        // check batch hash
+        uint256 _batchIndex = BatchHeaderCodecV0.getBatchIndex(memPtr);
+        require(committedBatches[_batchIndex] == _batchHash, "incorrect batch
↪  hash");

         require(lastFinalizedBatchIndex < batchIndex, "batch already finalized");
         require(committedBatches[batchIndex] != 0, "batch not exist");
         require(challenges[batchIndex].challenger == address(0), "batch already
↪  challenged");

         // check challenge window
         require(batchInsideChallengeWindow(batchIndex), "cannot challenge batch
↪  outside the challenge window");
         // check challenge amount
         require(msg.value >= IL1Staking(l1StakingContract).challengeDeposit(),
↪  "insufficient value");

         batchChallenged = batchIndex;
         challenges[batchIndex] = BatchChallenge(batchIndex, _msgSender(),
↪  msg.value, block.timestamp, false, false);
         emit ChallengeState(batchIndex, _msgSender(), msg.value);
```

```
        for (uint256 i = lastFinalizedBatchIndex + 1; i <= lastCommittedBatchIndex;
↪   i++) {
            if (i != batchIndex) {
                batchDataStore[i].finalizeTimestamp += proofWindow;
            }
        }

        inChallenge = true;
    }
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/morph-l2/morph/pull/558

## Issue M-7: Delegators can lose their rewards when a delegator has removed a delegatee and claims all of his rewards before delegating again to a previous removed delegatee.

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/157

### Found by

0xStalin

### Summary

The lose of funds is the result of two bugs, which are:

1. The first bug occurs when **a delegator claims all his rewards after he removed one or more delegatees.** This first bug is about the logic implemented to claim all the delegator's rewards. When claiming all the rewards after a delegator has removed a delegatee, the undelegated delegatee is removed from the AddressSet of delegatees.

- The problem is that when a delegatee is removed, the length of the AddressSet is reduced, which it affects the total number of delegatees that will be iterated to claim the rewards. As the AddressSet shrinks, the `i` variable controlling the iterations of the for loop grows, which at some point, `i` will be equals to the new length of the shrinked AddressSet, causing the for to exit the iteration and leaving some delegatees without claiming their rewards.

```
function claimAll(address delegator, uint256 targetEpochIndex) external
↪  onlyL2StakingContract {
    ...
    //@audit-issue => `i` grows in each iteration while the delegatees AddressSet
↪  shrinks each time an undelegated delegatee is foud!
    for (uint256 i = 0; i < unclaimed[delegator].delegatees.length(); i++) {
        address delegatee = unclaimed[delegator].delegatees.at(i);
        if (
            unclaimed[delegator].delegatees.contains(delegatee) &&
            unclaimed[delegator].unclaimedStart[delegatee] <= endEpochIndex
        ) {
            ///@audit => If the delegator has unclaimed the delegatee at an epoch <
↪  endEpochIndex, the delegatee will be removed from the delegatees AddressSet
```

```
                //@audit-issue => Removing the delegatee from the AddressSet causes
    ↪   the Set to shrink!
                reward += _claim(delegatee, delegator, endEpochIndex);
            }
        }
        ...
}

function _claim(address delegatee, address delegator, uint256 endEpochIndex)
    ↪   internal returns (uint256 reward) {
        ...

        for (uint256 i = unclaimed[delegator].unclaimedStart[delegatee]; i <=
    ↪   endEpochIndex; i++) {
            ...

            // if undelegated, remove delegator unclaimed info after claimed all
            if (unclaimed[delegator].undelegated[delegatee] &&
    ↪   unclaimed[delegator].unclaimedEnd[delegatee] == i) {
                //@audit-issue => Removing the delegatee from the AddressSet causes the
    ↪   Set to shrink!
                unclaimed[delegator].delegatees.remove(delegatee);
                delete unclaimed[delegator].undelegated[delegatee];
                delete unclaimed[delegator].unclaimedStart[delegatee];
                delete unclaimed[delegator].unclaimedEnd[delegatee];
                break;
            }
        }
        ...
}
```

Find below a simple PoC where it is demonstrated this bug. It basically adds and undelegates some delegatees to one delegator, then, the delegator claims all of his rewards, and, because the delegator has some undelegated delegatees, the bug will occur causing the function to not claim the rewards of all the delegatees.

- Create a new file on the test/ folder and add the below code in it:

```
pragma solidity ^0.8.13;

import {EnumerableSetUpgradeable} from "@openzeppelin/contracts-upgradeable/utils/s
    ↪   tructs/EnumerableSetUpgradeable.sol";

import {Test, console2} from "forge-std/Test.sol";

contract SimpleDelegations {

  using EnumerableSetUpgradeable for EnumerableSetUpgradeable.AddressSet;

  struct Unclaimed {
      EnumerableSetUpgradeable.AddressSet delegatees;
      mapping(address delegator => bool undelegated) undelegated;
      // mapping(address delegator => uint256 startEpoch) unclaimedStart;
      // mapping(address delegator => uint256 endEpoch) unclaimedEnd;
  }

  mapping(address delegator => Unclaimed) private unclaimed;

  function addDelegatee(address delegatee) external {
    unclaimed[msg.sender].delegatees.add(delegatee);
  }

  function undelegate(address delegatee) external {
    unclaimed[msg.sender].undelegated[delegatee] = true;
  }

  //@audit-issue => Existing logic that causes to not claim all the delegatees
  //@audit-issue => The problem is that when there are undelegated delegatees, they
    ↪  will be removed from the delegator in the `_claim()`, which makes the
    ↪  `unclaimed.delegatees` AddressSet lenght to shrink, which unintentionally
    ↪  causes the for loop to do less iterations than the original amount of
    ↪  delegatees at the begining of the call!
  function claimAll() external returns (uint256 reward) {
    for (uint256 i = 0; i < unclaimed[msg.sender].delegatees.length(); i++) {
      console2.log("delegatees.lenght: ",
 ↪  unclaimed[msg.sender].delegatees.length());
      address delegatee = unclaimed[msg.sender].delegatees.at(i);
      // console2.log("i: ", i);
      console2.log("delegatee: ", delegatee);
      if (
          unclaimed[msg.sender].delegatees.contains(delegatee)
          // unclaimed[delegator].unclaimedStart[delegatee] <= endEpochIndex
      ) {
          reward += _claim(delegatee, msg.sender);
      }

    }
  }

  //@audit-recommendation => This would be the fix for the first bug!
  // function claimAll() external returns (uint256 reward) {
  //   uint256 totalDelegatees = unclaimed[msg.sender].delegatees.length();
  //   address[] memory delegatees = new address[](totalDelegatees);

  //   //@audit => Make a temporal copy of the current delegates of the delegator
  //   for (uint256 i = 0; i < unclaimed[msg.sender].delegatees.length(); i++) {
  //       delegatees[i] = unclaimed[msg.sender].delegatees.at(i);
  //   }
```

```
  //   //@audit => Iterate over the temporal copy, in this way, if a delegatee is
↪  removed from the AddressSet, the for loop will still iterate over all the
↪  delegatees at the start of the call!
  //   for (uint256 i = 0; i < delegatees.length; i++) {
  //     // console2.log("delegatee: ", delegatees[i]);
  //     if (
  //         unclaimed[msg.sender].delegatees.contains(delegatees[i])
  //         // unclaimed[delegator].unclaimedStart[delegatees[i]] <= endEpochIndex
  //     ) {
  //         reward += _claim(delegatees[i], msg.sender);
  //     }
  //   }
  // }

  function _claim(address delegatee, address delegator) internal returns(uint256
↪  reward) {
    require(unclaimed[delegator].delegatees.contains(delegatee), "no remaining
↪  reward");

    reward = 10;

    //@audit-issue => Removes an undelegated delegatee from the delegator's
↪  delegatees!
    if (unclaimed[delegator].undelegated[delegatee]) {
      //@audit-issue => The `unclaimed[delegator].delegatees.length()` shrinks,
↪  causing the for loop to not iterate over all the delegatees!
      unclaimed[delegator].delegatees.remove(delegatee);
      delete unclaimed[delegator].undelegated[delegatee];
    }
  }

  function getDelegatees() external view returns (address[] memory) {
    return unclaimed[msg.sender].delegatees.values();
  }
}

contract BugWhenClaimingAllRewards is Test {
  function test_claimingAllRewardsReproducingBug() public {
    SimpleDelegations delegations = new SimpleDelegations();

    delegations.addDelegatee(address(1));
    delegations.addDelegatee(address(2));
    delegations.addDelegatee(address(3));

    delegations.undelegate(address(1));
    delegations.undelegate(address(3));

    //@audit => Should claim 30 of rewards! There are 3 delegatees and each
↪  delegatee gives 10 of rewards
    uint256 rewards = delegations.claimAll();
    console2.log("Total rewards: ", rewards);

    console2.log("delegatees list after claiming");
    address[] memory delegatees = delegations.getDelegatees();
    for(uint i = 0; i < delegatees.length; i++) {
      console2.log("delegatee: ", delegatees[i]);
    }

    //@audit => If all delegatees would have been claimed, then, delegatees 1 & 3
↪   would have been deleted from the `unclaimed.delegatees` AddressSet, but because
↪   the delegatee 3 was never reached, (and the rewards of this delegatee were not
↪   claimed), this delegatee is still part of the list of delegatees for the
↪   delegator with unclaimed rewards!
  }
}
```

Run the previous PoC with the next command: `forgetest--match-testtest_claimingAllRewardsReproducingBug-vvvv` @note => This PoC simply reproduces the first bug. The next PoC will be a full coded PoC working with the contracts of the system where the full scenario that causes the users to lose their rewards is reproduced.

2. The second bug occurs when **a delegator delegates to a previously undelegated delegatee.** When delegating to a previous undelegated delegatee, the logic does not check if there are any pending rewards of this delegatee to be claimed, it simply updates the  to the effectiveEpoch.

```
function notifyDelegation(
    ...
    //@audit => effectiveEpoch would be the next epoch (currentEpoch() + 1)
    uint256 effectiveEpoch,
    ...
) public onlyL2StakingContract {
  ...
  // update unclaimed info
  if (newDelegation) {
      unclaimed[delegator].delegatees.add(delegatee);
      //@audit => Regardless if there is any unclaimed rewards for this delegatee,
↪   sets the `unclaimedStart` to be the effectiveEpoch.
      unclaimed[delegator].unclaimedStart[delegatee] = effectiveEpoch;
  }
}
```

Setting the `unclaimedStart` without verifying if there are any pending rewards causes that rewards can only be claimed starting at the effectiveEpoch, any pending rewards that were earned prior to the effectiveEpoch will be lost, since the enforces that the epoch been claimed must not be <= the unclaimedStart.

```
function _claim(address delegatee, address delegator, uint256 endEpochIndex)
↪   internal returns (uint256 reward) {
  require(unclaimed[delegator].delegatees.contains(delegatee), "no remaining
↪   reward");
  //@audit-info => Rewards can be claimed only from the `unclaimedStart` epoch
↪   onwards!
  require(unclaimed[delegator].unclaimedStart[delegatee] <= endEpochIndex, "all
↪   reward claimed");
}
```

As we will see in the next coded PoC, the user did everything correctly, he claimed all of his rewards, but due to the first bug, some rewards were left unclaimed. And then, the user claims his undelegation for an undelegated delegatee, afterwards, he proceeded to delegate a lower amount to an undelegated delegatee. The combination of the two bugs is what causes the lose of the rewards that were not claimed when the user indicated to claim all of his rewards. The user did not make any misstake, the bugs present on the contract caused the user to lose his rewards.

## Root Cause

Bug when claiming all delegator's rewards after a delegatee(s) were removed.

Bug when a delegator delegatees to a delegatee that was previously undelegated.

## Internal pre-conditions

User claims all his rewards after one or more delegatees were removed, and afterwards, it delegates to a delegatee that was previously removed.

## External pre-conditions

none

## Attack Path

1. Delegator undelegates to one or more of his delegatees.
2. Delegator claims all of his rewards.
3. Delegator claims his undelegations on the L2Staking contract.
4. Delegator delegates again to a previous removed delegatee.

## Impact

User's rewards are lost and become unclaimable, those rewards get stuck on the Distribute contract.

## PoC

Add the next PoC in the  test file:

```
function test_DelegatorLosesRewardsPoC() public {
    hevm.startPrank(alice);
    morphToken.approve(address(l2Staking), type(uint256).max);
    l2Staking.delegateStake(firstStaker, 5 ether);
    l2Staking.delegateStake(secondStaker, 5 ether);
    hevm.stopPrank();

    //@audit => Bob delegates to the 3 stakers
    hevm.startPrank(bob);
    morphToken.approve(address(l2Staking), type(uint256).max);
    l2Staking.delegateStake(firstStaker, 5 ether);
    l2Staking.delegateStake(secondStaker, 5 ether);
    l2Staking.delegateStake(thirdStaker, 5 ether);
    hevm.stopPrank();

    uint256 time = REWARD_EPOCH;
    hevm.warp(time);

    hevm.prank(multisig);
    l2Staking.startReward();

    // staker set commission
    hevm.prank(firstStaker);
    l2Staking.setCommissionRate(1);
    hevm.prank(secondStaker);
    l2Staking.setCommissionRate(1);
    hevm.prank(thirdStaker);
    l2Staking.setCommissionRate(1);

    // *************** epoch = 1 ******************** //
    time = REWARD_EPOCH * 2;
    hevm.warp(time);

    uint256 blocksCountOfEpoch = REWARD_EPOCH / 3;
    hevm.roll(blocksCountOfEpoch * 2);
    hevm.prank(oracleAddress);
    record.setLatestRewardEpochBlock(blocksCountOfEpoch);
    _updateDistribute(0);

    // *************** epoch = 2 ******************** //
    time = REWARD_EPOCH * 3;
    hevm.roll(blocksCountOfEpoch * 3);
    hevm.warp(time);
    _updateDistribute(1);

    // *************** epoch = 3 ******************** //
    time = REWARD_EPOCH * 4;
    hevm.roll(blocksCountOfEpoch * 4);
    hevm.warp(time);
    _updateDistribute(2);

    uint256 bobReward1;
    uint256 bobReward2;
    uint256 bobReward3;

    {
        (address[] memory delegetees, uint256[] memory bobRewards) =
↪   distribute.queryAllUnclaimed(bob);
        bobReward1 = distribute.queryUnclaimed(firstStaker, bob);
        bobReward2 = distribute.queryUnclaimed(secondStaker, bob);
        bobReward3 = distribute.queryUnclaimed(thirdStaker, bob);
        assertEq(delegetees[0], firstStaker);
        assertEq(delegetees[1], secondStaker);
        assertEq(delegetees[2], thirdStaker);
        assertEq(bobRewards[0], bobReward1);
        assertEq(bobRewards[1], bobReward2);
        assertEq(bobRewards[2], bobReward3);
    }

    // *************** epoch = 4 ******************** //
    time = REWARD_EPOCH * 5;
    hevm.roll(blocksCountOfEpoch * 5);
    hevm.warp(time);
    _updateDistribute(3);

    //@audit => Bob undelegates to 1st and 3rd staker at epoch 4 (effective epoch
↪   5)!
    hevm.startPrank(bob);
    l2Staking.undelegateStake(firstStaker);
    l2Staking.undelegateStake(thirdStaker);
    // l2Staking.undelegateStake(secondStaker);
    IL2Staking.Undelegation[] memory undelegations =
↪   l2Staking.getUndelegations(bob);
    assertEq(undelegations.length, 2);

    // *************** epoch = 5 ******************** //
    time = REWARD_EPOCH * 6;
    hevm.roll(blocksCountOfEpoch * 6);
    hevm.warp(time);
    _updateDistribute(4);

    //@audit => Undelegation happened on epoch 4 (effective epoch 5)
    time = rewardStartTime + REWARD_EPOCH * (ROLLUP_EPOCH + 5);
    hevm.warp(time);

    bobReward1 = distribute.queryUnclaimed(firstStaker, bob);
    bobReward2 = distribute.queryUnclaimed(secondStaker, bob);
    bobReward3 = distribute.queryUnclaimed(thirdStaker, bob);


    //@audit => Bob claims all of his rewards for all the epochs
    hevm.startPrank(bob);
    uint256 balanceBefore = morphToken.balanceOf(bob);
    l2Staking.claimReward(address(0), 0);

    uint256 balanceAfter = morphToken.balanceOf(bob);
    //@audit => assert that balanceAfter is less than balanceBefore + all the
↪   rewards that Bob has earned.
    assert(balanceAfter < balanceBefore + bobReward1 + bobReward2 + bobReward3);

    //@audit => Bob claims his undelegations on the L2Staking
    l2Staking.claimUndelegation();

    //@audit => Rewards that Bob can claim before delegating to an staker that he
↪   undelegeated in the past
    //@audit => This value will be used to validate that Bob will lose rewards
↪   after he delegates to the 3rd staker (because the bug when claiming all the
↪   rewards while the delegator has undelegated delegatees), in combination with
↪   the bug when delegating to a delegatee that the user has pending rewards to
↪   claim.
    uint256 bobRewardsBeforeLosing;
    {
        (address[] memory delegetees, uint256[] memory bobRewards) =
↪   distribute.queryAllUnclaimed(bob);
        for(uint i = 0; i < bobRewards.length; i++) {
            bobRewardsBeforeLosing += bobRewards[i];
        }
    }
    //@audit => Substracting 1 ether because that will be the new delegation for
↪   the thirdStaker
    uint256 bobMorphBalanceBeforeDelegating = morphToken.balanceOf(bob) - 1 ether;

    //@audit => Bob delegates again to 3rd staker a lower amount than the previous
↪   delegation
    l2Staking.delegateStake(thirdStaker, 1 ether);

    //@audit => Bob's rewards after delegating to the 3rd staker.
    uint256 bobRewardsAfterLosing;
    {
        (address[] memory delegetees, uint256[] memory bobRewards) =
↪   distribute.queryAllUnclaimed(bob);
        for(uint i = 0; i < bobRewards.length; i++) {
            bobRewardsAfterLosing += bobRewards[i];
        }
    }
    uint256 bobMorphBalanceAfterDelegating = morphToken.balanceOf(bob);
    //@audit-issue => This assertion validates that Bob lost rewards after the
 ↪  delegation to the 3rd staker.
    assert(bobRewardsBeforeLosing > bobRewardsAfterLosing &&
 ↪  bobMorphBalanceBeforeDelegating == bobMorphBalanceAfterDelegating);
}
```

Run the previous PoC with the next command: `forgetest--match-testtest_DelegatorLosesRewardsPoC-vvvv`

## Mitigation

Since there are two bugs involved in this problem, it is required to fix each of them.

The mitigation for the first bug (problem when claiming all the rewards):

- Refactor the  as follows:

```
function claimAll(address delegator, uint256 targetEpochIndex) external returns
 ↪  (uint256 reward) {
  require(mintedEpochCount != 0, "not minted yet");
  uint256 endEpochIndex = (targetEpochIndex == 0 || targetEpochIndex >
 ↪  mintedEpochCount - 1)
      ? mintedEpochCount - 1
      : targetEpochIndex;
  uint256 reward;

  //@audit => Make a temporal copy of the current delegates of the delegator
  uint256 totalDelegatees = unclaimed[delegator].delegatees.length();
  address[] memory delegatees = new address[](totalDelegatees);

  for (uint256 i = 0; i < unclaimed[delegator].delegatees.length(); i++) {
        delegatees[i] = unclaimed[delegator].delegatees.at(i);
  }

  //@audit => Iterate over the temporal copy, in this way, if a delegatee is
 ↪  removed from the AddressSet, the for loop will still iterate over all the
 ↪  delegatees at the start of the call!
  for (uint256 i = 0; i < delegatees.length; i++) {
    if (
          unclaimed[delegator].delegatees.contains(delegatees[i]) &&
              unclaimed[delegator].unclaimedStart[delegatee] <= endEpochIndex
    ) {
          reward += _claim(delegatees[i], delegator);
    }
  }

  if (reward > 0) {
      _transfer(delegator, reward);
  }
}
```

The mitigation for the second bug (delegating to a previous undelegated delegatee causes all pending rewards to be lost):

- Before setting the , check if there is any pending rewards, and if there are any, either revert the tx or proceed to claim the rewards of the delegatee on behalf of the delegator.

```
function notifyDelegation(
    ...
) public onlyL2StakingContract {
  ...
  // update unclaimed info
  if (newDelegation) {
      //@audit-recommendation => Add the next code to validate that the delegator
  ↪  does not have any pending rewards to claim on the delegatee!
      if(unclaimed[delegator].undelegated[delegatee] &&
  ↪  unclaimed[delegator].unclaimedEnd[delegatee] != 0) {
        //@audit => The claim function will stop claiming rewards when it reaches
  ↪  the epoch when the delegator undelegated the delegatee!
        uint256 rewards = _claim(delegatee, delegator, effectiveEpoch - 1);
        if (reward > 0) {
            _transfer(delegator, reward);
        }
      }

      unclaimed[delegator].delegatees.add(delegatee);
      unclaimed[delegator].unclaimedStart[delegatee] = effectiveEpoch;
  }
}
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits: https://github.com/morph-l2/morph/pull/594

## Issue M-8: Stakers lose their commission if they unstake as they cannot claim their pending rewards anymore after unstaking

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/168

### Found by

0xStalin, mstpr-brainbot, n4nika, p0wd3r, sammy, underdog, zraxx

### Summary

Once a staker unstakes or gets removed, they permanently lose access to all their accrued commissions.

This is a problem as it can either happen mistakenly or if the staker gets removed by an admin or slashed. However even in that case, they should still be able to claim their previously accrued rewards since they did not act negatively during that period.

### Root Cause

has the `onlyStaker` modifier, making it only callable by stakers.

### Internal pre-conditions

None

### External pre-conditions

None

### Attack Path

Issue path in this case

- Staker stakes and accrues commissions over a few epochs
- Now either the staker unstakes or gets removed forcibly by the admin
- The staker has now lost access to all previously accrued rewards

## Impact

Loss of unclaimed rewards

## PoC

*No response*

## Mitigation

Consider removing the `onlyStaker` modifier and allow anyone to call it. This should not be a problem since normal users do not have any claimable commission anyways except if they were a staker before.

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/morph-l2/morph/pull/516

## Issue M-9: Attacker can fill merkle tree in `L2ToL1MessagePasser`, blocking any future withdrawals

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/178

The protocol has acknowledged this issue.

### Found by

mstpr-brainbot, n4nika

### Summary

An attacker can fill the merkle tree used for withdrawals from `L2->L1`, preventing any withdrawals from `L2` to `L1`.

### Root Cause

The protocol uses one single merkle tree with a maximum of `2**32-1` entries for all ever happening withdrawals. Once that tree is full, any calls made to `L2CrossDomainMessenger.sol::_sendMessage` will fail, since , called in `L2ToL1MessagePasser.sol::appendMessage` will revert.

```
function _appendMessageHash(bytes32 leafHash) internal {
    bytes32 node = leafHash;

    // Avoid overflowing the Merkle tree (and prevent edge case in computing
 ↪  `_branch`)
    if (leafNodesCount >= _MAX_DEPOSIT_COUNT) {
        revert MerkleTreeFull();
    }
    // [...]
}
```

### Internal pre-conditions

None

### External pre-conditions

None

## Attack Path

- attacker deploys a contract with a function which initiates `200` withdrawals from `L2` to `L1` per transaction by calling `l2CrossDomainMessenger.sendMessage(payable(0),0,"",0)` `200` times
- they then automate calling that contract as many times as it takes to fill the `2**32-1` entries of the withdrawal merkel tree in `Tree.sol` ( `21_000_000times`)
- this fills the merkle tree and once it is full, any withdrawals are blocked permanently

Cost for a DoS with the lowest gas cost: `51152USD` at `2400USD/ETH`

Note that this exploit takes some time to execute. However with the low gas costs and block times on L2, it is absolutely feasible to do so causing massive damage to the protocol. Additionally, if the ether price goes down, this will cost even less to execute.

## Impact

Permanent DoS of the rollup as no `L2->L1` withdrawals/messages will be possible anymore.

## PoC

The following exploit puts `200calls` into one transaction, using close to the block gas limit of `10_000_000gas`. To run it please add it to `L2Staking.t.sol` and execute it with `forge test --match-test test_lowGasSendMessage`.

```
function test_lowGasSendMessage() public {
    for (uint256 i = 0; i < 200; ++i) {
        l2CrossDomainMessenger.sendMessage(payable(0), 0, "", 0);
    }
}
```

The forge output will show it costing about `9_924_957gas` with which the calculations in `AttackPath` were made.

## Mitigation

Consider adding different handling for when a merkle tree is full, not bricking the rollup because of a filled merkle tree.

## Issue M-10: Malicious sequencer can DoS proposal execution by inflating the amount of proposals to be pruned

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/186

### Found by

HaxSecurity, underdog

### Summary

A malicious sequencer can inflate the amount of proposals in `Gov.sol` in order to force the previous proposals invalidation process in `_executeProposal()` to run out of gas, making it impossible to execute proposals.

### Root Cause

Found in Gov.sol#L257-L260.

The `Gov.sol` contract incorporates a "proposal pruning process". Every time a proposal is executed, all the previous proposals since `undeletedProposalStart` will be pruned (even if they are executed or not). This is done in order to avoid a situation where older proposals get executed, overriding the governance values of newly executed proposals.

As shown in the following code snippet, the "pruning process" works by iterating all proposals since the latest `undeletedProposalStart`, and until the current proposal ID. It is also worth noting that `undeletedProposalStart` is only updated **when executing a proposal,** and is set to the ID of the proposal being executed so that next proposal executions don't need to prune from the first proposal ID:

```
// Gov.sol

function _executeProposal(uint256 proposalID) internal {
        ...

        // when a proposal is passed, the previous proposals will be invalidated
 ↪    and deleted
        for (uint256 i = undeletedProposalStart; i < proposalID; i++) {
            delete proposalData[i];
            delete proposalInfos[i];
            delete votes[i];
        }
        undeletedProposalStart = proposalID;

        ...
}
```

However, this iteration can lead to an out-of-gas error, as a malicious sequencer can inflate the amount of proposal ID's due to the lack of limits in proposal creations:

```
// Gov.sol
function createProposal(ProposalData calldata proposal) external onlySequencer
↪   returns (uint256) {
        require(proposal.rollupEpoch != 0, "invalid rollup epoch");
        require(proposal.maxChunks > 0, "invalid max chunks");
        require(
            proposal.batchBlockInterval != 0 || proposal.batchMaxBytes != 0 ||
↪   proposal.batchTimeout != 0,
            "invalid batch params"
        );

        currentProposalID++;
        proposalData[currentProposalID] = proposal;
        proposalInfos[currentProposalID] = ProposalInfo(block.timestamp +
↪   votingDuration, false);

        emit ProposalCreated(
            currentProposalID,
            _msgSender(),
            proposal.batchBlockInterval,
            proposal.batchMaxBytes,
            proposal.batchTimeout,
            proposal.maxChunks,
            proposal.rollupEpoch
        );

        return (currentProposalID);
    }
```

As shown in the snippet, any sequencer can create a proposal, and there's no limit to proposal creation, allowing the proposal inflation scenario to occur.

## Internal pre-conditions

- A staker in Morph needs to be a sequencer in the sequencer set (achieved by obtaining delegations, or by simply being added as a staker in the `L2Staking` contract if rewards have not started)
- The sequencer needs to call `createProposal()` several times so the amount of proposals created is inflated, and the `currentProposalID` tracker becomes a

number big enough to DoS proposal pruning

## External pre-conditions

None.

## Attack Path

1. A staker stakes in `L1Staking`. A cross-layer message is sent from L1 to L2, and the staker is registered to the `L2Staking` contract as a staker.
2. If rewards have not started and the sequencer set is not filled, the staker will be directly added to the sequencer set, becoming a sequencer. Otherwise, the staker needs to have some MORPH tokens delegated so that he can become a sequencer.
3. After becoming a sequencer, `createProposal()` is called several times to inflate the amount of proposal ID's. The sequencer can set unreasonable values so that voting and executing any of the proposals leads the system to behave dangerously, making it disappealing for sequencers to vote for any of the proposals and avoid the OOG issue by executing certain proposals.
4. Because `currentProposalID` has been inflated, any proposal created after the inflation will try to prune all the previous fake proposals, leading to a DoS and runnning out of gas. This will effectively DoS proposal creation.

## Impact

Sequencers won't be able to execute proposals in the governance contract, making it impossible to update global network parameters used by Morph.

## PoC

*No response*

## Mitigation

Implement a cooldown period between proposal creations. This will prevent creating a huge amount of proposals, as the attacker will need to wait for the cooldown period to finish in order to create another proposal. If the owner identifies suspicious behavior, he can then remove him in the L1 staking contract, and effectively removing the attacker as sequencer in the L2.

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/morph-l2/morph/pull/603

## Issue M-11: In the `revertBatch` function, `inChallenge` is set to `false` incorrectly, causing challenges to continue after the protocol is paused.

Source: https://github.com/sherlock-audit/2024-08-morphl2-judging/issues/220

### Found by

0xRajkumar, mstpr-brainbot, p0wd3r, ulas

### Summary

An unchecked batch reversion will cause challenge invalidation for any committed batch, leading to batch rollback issues for challengers, as the isChallenged flag will reset unexpectedly.

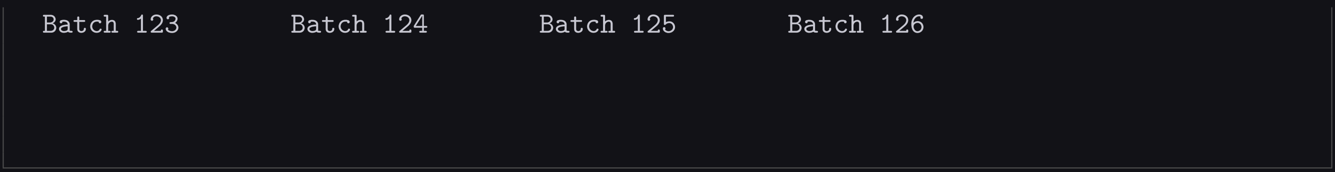
### Root Cause

In the revertBatch function, the `inChallenge` state is set to false even if the batch that was challenged is not part of the reverted batch set. This causes ongoing challenges to be incorrectly invalidated:

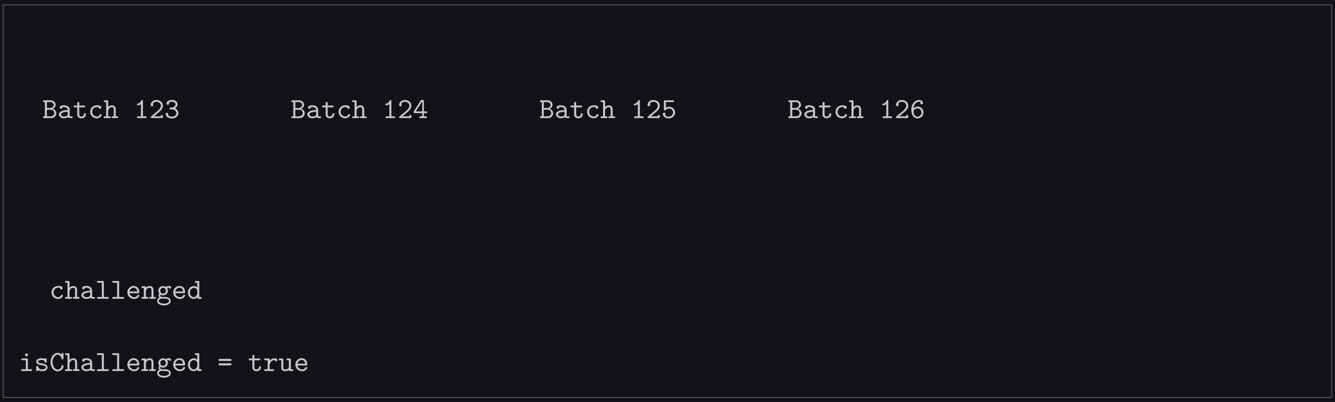
```
function revertBatch(bytes calldata _batchHeader, uint256 _count) external
↪  onlyOwner {
            .... REDACTED FOR BREVITY ...
            if (!challenges[_batchIndex].finished) {
                batchChallengeReward[challenges[_batchIndex].challenger] +=
↪  challenges[_batchIndex].challengeDeposit;
                inChallenge = false;
            }
            .... REDACTED FOR BREVITY ...
```

In the above code, `if(!challenges[_batchIndex].finished)` will hold `true` for challenges that doesn't exist. If there are no challenges for a specific `_batchIndex`, then `challenges[_batchIndex].finished` will be `false` which in turn will make the `if` condition true.

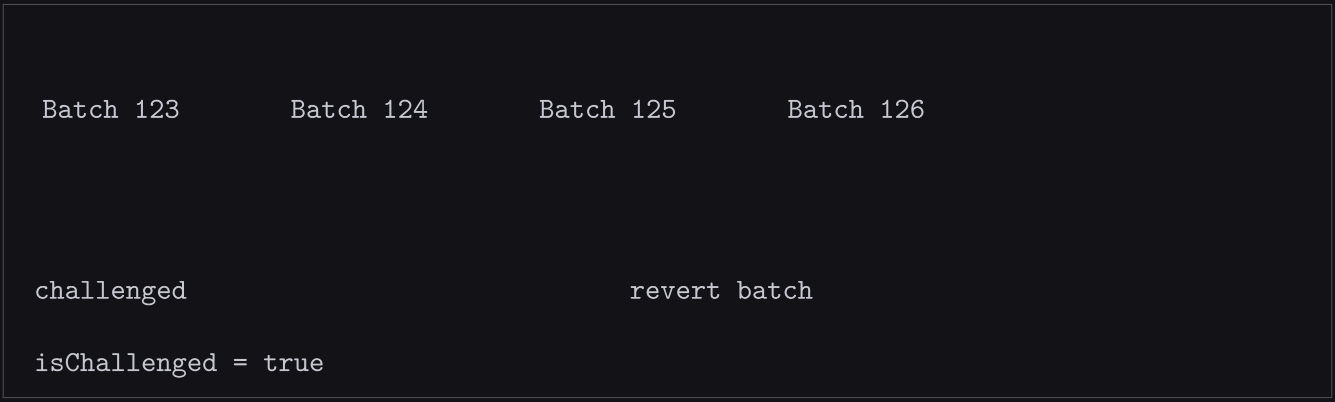
This will cause `inChallenge` to be set to `false` even when there are ongoing challenges. Lets assume the following batches were commit to L1 Rollup::



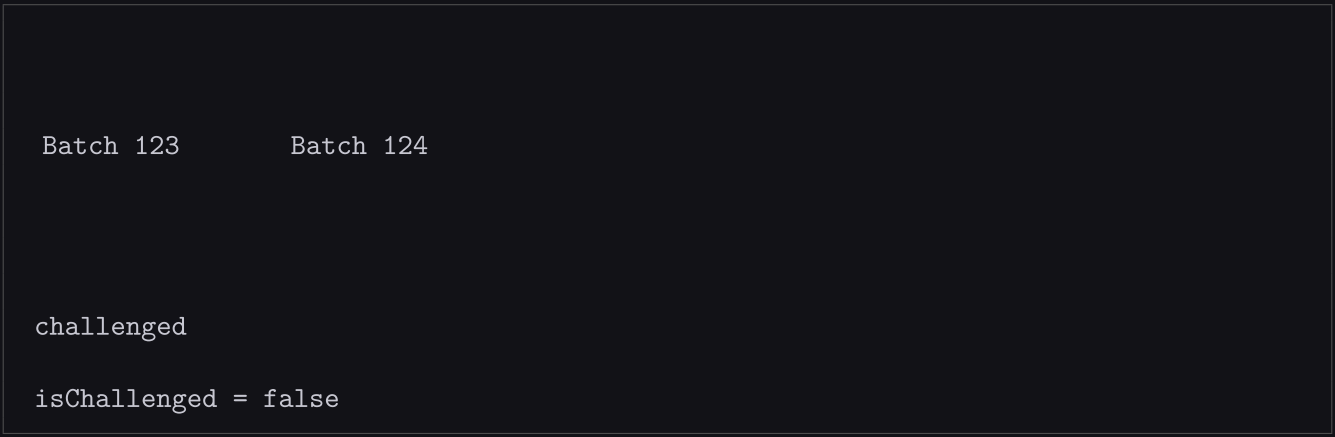
Challenger calls `challengeState` on batch 123. This sets `isChallenged` storage variable to true.



While the challenge is ongoing owner calls `revertBatch` on Batch 125 to revert both Batch 125 and Batch 126.



Due to the bug in the `revertBatch` function, `isChallenged` is set to `false` even though the challenged batch wasn't in the reverted batches.



This will lead to issues when the protocol is paused. Due to the following check in the `set`

`Pause` function, the challenge will not be deleted while the protocol is paused:

```
function setPause(bool _status) external onlyOwner {
            .... REDACTED FOR BREVITY ...
            // inChallenge is set to false due to the bug in revertBatch
             if (inChallenge) {
                batchChallengeReward[challenges[batchChallenged].challenger] +=
↪   challenges[batchChallenged]
                    .challengeDeposit;
                delete challenges[batchChallenged];
                inChallenge = false;
            }
            .... REDACTED FOR BREVITY ...
```

During the protocol pause, the prover will not be able to verify the proof and if the pause period is larger than the proof window, prover will lose the challenge and gets slashed.

## Internal pre-conditions

1. Owner calls `revertBatch` on batch n, reverting the nth batch.
2. Challenger monitors the mempool and initiates a challenge on the n-1 batch.
3. Due to the bug in `revertBatch`, the `inChallenge` flag is reset to `false`, even though batch n-1 is under challenge.
4. Owner calls `setPause` and the protocol is paused longer than the challenge window.

## External pre-conditions

*No response*

## Attack Path

1. The owner calls `revertBatch` on batch n, reverting batch n.
2. A challenger monitors the mempool and calls `challengeBatch` on batch n-1.
3. The `revertBatch` function incorrectly resets the `inChallenge` flag to false despite batch n-1 being under challenge.
4. The protocol is paused, preventing the challenge from being deleted.
5. The prover cannot prove the batch in time due to the paused protocol.
6. The prover gets slashed, even though the batch is valid.

## Impact

If the protocol is paused when there's an ongoing challenge (albeit inChallenge is set to `false` due to the vulnerability explained above) , the protocol slashes the batch submitter for failing to prove the batch within the challenge window, even though the batch is valid. The challenger may incorrectly receive the challenge reward + slash reward despite no actual issue in the batch.

## PoC

*No response*

## Mitigation

Use `batchInChallenge` function to verify the batch is indeed challenged:

```
function revertBatch(bytes calldata _batchHeader, uint256 _count) external
↪ onlyOwner {
            // ... Redacted for brevity ...
        while (_count > 0) {
            emit RevertBatch(_batchIndex, _batchHash);

            committedBatches[_batchIndex] = bytes32(0);
            // if challenge exist and not finished yet, return challenge deposit to
↪ challenger
            if (batchInChallenge(_batchIndex)) {
                batchChallengeReward[challenges[_batchIndex].challenger] +=
↪ challenges[_batchIndex].challengeDeposit;
                inChallenge = false;
            }
            delete challenges[_batchIndex];

                    // ... Redacted for brevity ...
        }
    }
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/morph-l2/morph/pull/561

## Disclaimers