Subsquid Security Review

Pashov Audit Group

Conducted by: said, carrotsmuggler, SpicyMeatball

January 29th 2024 - February 4th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **subsquid-network-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Subsquid Network

Subsquid is a protocol for managing nodes participating in the distributed query and data lake network, and distributing rewards between them.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash **- 3545236f5b34076820fe747eb607a16d8b664a08**

fixes review commit hash **- 4bcd8ee2f8de17d0877bba876574309fcaf227e1**

Scope

The following smart contracts were in scope of the audit:

- `arbitrum/tSQD`
- `gateway-strategies/EqualStrategy`
- `AccessControlledPausable`
- `DistributedRewardsDistribution`
- `Executable`
- `GatewayRegistry`
- `NetworkController`
- `RewardCalculation`
- `RewardTreasury`
- `Router`
- `Staking`
- `TemporaryHolding`
- `TemporaryHoldingFactory`
- `Vesting`
- `VestingFactory`
- `WorkerRegistration`
- `tSQD`
- `interfaces/**`

7. Executive Summary

Over the course of the security review, said, carrotsmuggler, SpicyMeatball engaged with Subsquid to review Subsquid Network. In this period of time a total of **15** issues were uncovered.

Protocol Summary

Protocol Name	Subsquid Network
Repository	https://github.com/subsquid/subsquid-network-contracts
Date	January 29th 2024 - February 4th 2024
Protocol Type	Distributed query engine

Findings Count

Severity	**Amount**
Critical	1
High	4
Medium	5
Low	5
Total Findings	**15**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Gateway creator can steal all tokens from the GatewayRegistry	Critical	Resolved
[H-01]	Missing check on toBlock allows distributors to change past rewards	High	Resolved
[H-02]	computationUnitsAvailable can overestimate number of available units if staking duration is too small	High	Resolved
[H-03]	Workers could withdraw without deregister and waiting for the lock period	High	Resolved
[H-04]	Lack of access control on tSQD's registerTokenOnL2	High	Acknowledged
[M-01]	setGatewayAddress incorrectly updates address of the gateway	Medium	Resolved
[M-02]	activeWorkerIds have no size limit, can grow unbounded and gas-grief / cause OOG errors	Medium	Resolved
[M-03]	User can lock tokens from the TemporaryHolding for an "infinite" amount of time	Medium	Acknowledged
[M-04]	Protocol uses manipulatable randomness	Medium	Acknowledged
[M-05]	Vesting contract can lead to frequent reverts	Medium	Resolved
[L-01]	PUSH0 not supported on Arbitrum	Low	Resolved
[L-02]	If the bond amount ever increased, there is no direct way to update	Low	Acknowledged

	workers bond		
[L-03]	Worker is removed from the active workers list too early	Low	Resolved
[L-04]	deposit function in Staking does not check whether the worker is still active	Low	Resolved
[L-05]	Unsuccessful / overwritten commits cannot be approved again	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Gateway creator can steal all tokens from the GatewayRegistry

Severity

Impact: High, user's funds will be stolen

Likelihood: High, can be exploited by anyone and easy to implement

Description

`GatewayRegistry` contract allows users to register and stake tokens into gateways to receive computation units CUs. First, the user registers a gateway,

```
function register(
    bytescalldatapeerId,
    stringmemorymetadata,
    addressgatewayAddress
  ) public whenNotPaused {
    require(peerId.length > 0, "Cannot set empty peerId");
    bytes32 peerIdHash = keccak256(peerId);
    require(gateways[peerIdHash].operator == address
      (0), "PeerId already registered");

    gateways[peerIdHash] = Gateway({
      operator: msg.sender,
      peerId: peerId,
      strategy: defaultStrategy,
      ownAddress: gatewayAddress,
      metadata: metadata,
      totalStaked: 0,
>>    totalUnstaked: 0
    });
```

note `totalUnstaked` is set to 0. After this we can stake tokens

```
function _stakeWithoutTransfer
    (bytes calldata peerId, uint256 amount, uint128 durationBlocks) internal {
    (Gateway storage gateway, bytes32 peerIdHash) = _getGateway(peerId);
    _requireOperator(gateway);

    uint256 _computationUnits = computationUnitsAmount(amount, durationBlocks);
    uint128 lockStart = router.networkController().nextEpoch();
    uint128 lockEnd = lockStart + durationBlocks;
>>  stakes[peerIdHash].push(Stake
  (amount, _computationUnits, lockStart, lockEnd));
```

`stakes` mapping is used to track all user stakes. The problem arises when we unregister the gateway, we do not delete the `stakes`, it can be exploited in the following steps:

- unstake tokens from the gateway
- unregister gateway
- register new gateway with the same `peerId`
- since `totalUnstaked = 0`, we can unstake tokens again

```
function _unstakeable(Gateway storage gateway) internal view returns
    (uint256) {
    Stake[] memory _stakes = stakes[keccak256(gateway.peerId)];
    uint256 blockNumber = block.number;
    uint256 total = 0;
    for (uint256 i = 0; i < _stakes.length; i++) {
      Stake memory _stake = _stakes[i];
      if (_stake.lockEnd <= blockNumber) {
        total += _stake.amount;
      }
    }
    return total - gateway.totalUnstaked;
  }
```

Here is the coded POC for `GatewayRegistry.unstake.t.sol`

```
function test_StealStakes() public {
    uint256 amount = 100;
    address alice = address(0xA11cE);
    token.transfer(alice, amount);

    // stakers stake into their gateways
    gatewayRegistry.stake(peerId, amount, 200);
    vm.startPrank(alice);
    token.approve(address(gatewayRegistry), type(uint256).max);
    gatewayRegistry.register(bytes("alice"), "", address(0x6a7e));
    gatewayRegistry.stake(bytes("alice"), amount, 200);

    assertEq(token.balanceOf(address(gatewayRegistry)), 200);
    // exploit
    vm.roll(block.number + 300);
    gatewayRegistry.unstake(bytes("alice"), amount);
    gatewayRegistry.unregister(bytes("alice"));
    gatewayRegistry.register(bytes("alice"), "", address(0x6a7e));
    // unstake again
    gatewayRegistry.unstake(bytes("alice"), amount);
    assertEq(token.balanceOf(address(gatewayRegistry)), 0);
  }
```

Recommendations

Delete `stakes` mapping when gateway is being unregistered.

8.2. High Findings

[H-01] Missing check on `toBlock` allows distributors to change past rewards

Severity

Impact: High, Breaks sequential rewards, can reward same epoch multiple times

Likelihood: Medium, Requires malicious distributor/s

Description

The function `commit` in `DistributedRewardDistribution.sol` is used to save a reward commit. However this function does not check if the `toBlock` is larger than the `fromBlock`. Thus distributors can set `toBlock` to 0 even.

```
require(
    recipients.length == workerRewards.length,
    "Recipients and worker amounts length mismatch"
);
require(
    recipients.length == _stakerRewards.length,
    "Recipients and staker amounts length mismatch"
);

require(currentDistributor() == msg.sender, "Not a distributor");
//@audit missing check on block span
require(toBlock < block.number, "Future block");
```

This creates further problems in the `distribute` function itself. There is a check there which tries to force distributions to be sequential.

```
require(
    lastBlockRewarded == 0 || fromBlock == lastBlockRewarded + 1,
    "Not all blocks covered"
);
```

However, an user can set the `toBlock` to be lower than the `fromBlock` and this will break the sequence. In fact, if a user sets `toBlock` to 0, they can even set `lastBlockRewarded` to 0 since the assignment takes place in the next line.

```
lastBlockRewarded = toBlock;
```

POC:

The test `test_RunsDistributionAfter3Approves` can be modified with the blockspan running from 1 to 0 to demonstrate the issue.

```
function test_RunsDistributionAfter3Approves() public {
    (
        uint256[] memory recipients,
        uint256[] memory workerAmounts,
        uint256[] memory stakerAmounts
    ) = prepareRewards(1);
    rewardsDistribution.addDistributor(address(1));
    rewardsDistribution.addDistributor(address(2));
    rewardsDistribution.setApprovesRequired(3);
    vm.roll(10);
    rewardsDistribution.commit(
        1,
        0,
        recipients,
        workerAmounts,
        stakerAmounts
    );
    hoax(address(1));
    rewardsDistribution.approve(
        1,
        0,
        recipients,
        workerAmounts,
        stakerAmounts
    );
    hoax(address(2));
    rewardsDistribution.approve(
        1,
        0,
        recipients,
        workerAmounts,
        stakerAmounts
    );
}
```

Test passes with no issues.

Recommendations

Enforce the check

```
require(toBlock > fromBlock, "Invalid block span");
```

Either in the `commit` function or in the `distribute` function.

[H-02] `computationUnitsAvailable` can overestimate number of available units if staking duration is too small

Severity

Impact: High, can lead to inflated compute unit allocation

Likelihood: Medium, can be easily exploited by a malicious user at no cost

Description

The `computationUnitsAvailable` function computes the amount of computation units available to a `peerId`. This is defined in the `GatewayRegistry.sol` contract as shown below.

```
for (uint256 i = 0; i < _stakes.length; i++) {
    Stake memory _stake = _stakes[i];
    if (
        _stake.lockStart <= blockNumber && _stake.lockEnd > blockNumber
    ) {
        total +=
            (_stake.computationUnits * epochLength) /
            (uint256(_stake.lockEnd - _stake.lockStart));
    }
}
return total;
```

If `_stake.lockEnd - _stake.lockStart` is less than `epochLength`, the computation units available per epoch will be higher than the total amount of computation units available to the peerId.

The `_stake.computationUnits` is calculated during staking, and is the total amount of computation units available to the peerId during the entire staking duration. The objective of the `computationUnitsAvailable` function is to compute the amount of computation units available per block. However, if the staking duration is lower than the epoch length, the computation units available per block will be calculated to be higher than the total amount of computation units available to the peerId.

During staking, there is no restriction on the staking duration, so users can set it to be arbitrarily small. Thus `epochLength/duration` gives a large number, instead of calculating the inverse of the number of epochs passed.

A short POC demonstrates the issue:

```
function test_AttackStake() public {
    gatewayRegistry.stake(peerId, 10 ether, 1);
    GatewayRegistry.Stake[] memory stakes = gatewayRegistry.getStakes(
        peerId
    );
    goToNextEpoch();
    emit log_named_uint("Stake compute units", stakes[0].computationUnits);
    emit log_named_uint(
        "Available compute units",
        gatewayRegistry.computationUnitsAvailable(peerId)
    );
}
```

The output:

```
[PASS] test_AttackStake() (gas: 212925)
Logs:
  Stake compute units: 10
  Available compute units: 50
```

This shows that while the total number of units assigned was 10, the per-epoch units available is 50.

Since `computationUnitsAvailable` is used off-chain to calculate the amount of computation units to allocate to workers, this can be used to assign extra computational units than intended.

Thus malicious users can repeatedly stake small amounts and pump up the amount of available computation units, and then unstake to get back their stake. They can increase their computational units allocation by a factor of `epochLength` by setting the duration to 1.

Recommendations

Set a minimum staking duration of 1 epoch.

[H-03] Workers could withdraw without deregister and waiting for the lock period

Severity

Impact: High, this bypass the designed deregister flow and doesn't delete worker from active workers list.

Likelihood: Medium, as long as not currently active, worker can directly withdraw without deregister as a worker.

Description

The designed flow and expected worker's behavior for withdrawing their bond is to first deregister the registered `peerId`. After that, they have to wait for the lock period before they can trigger the `withdraw`, as it can be seen from `withdraw` functions expected flow.

```
/**
   * @dev Withdraws the bond of a worker.
   * @param peerId The unique peer ID of the worker.
   * @notice Worker must be inactive
   * @notice Worker must be registered by the caller
   * @notice Worker must be deregistered for at least lockPeriod // @audit -
   // prerequisite for withdraw
   */
  function withdraw(bytes calldata peerId) external whenNotPaused {
    uint256 workerId = workerIds[peerId];
    require(workerId != 0, "Worker not registered");
    Worker storage worker = workers[workerId];
    require(!isWorkerActive(worker), "Worker is active");
    require(worker.creator == msg.sender, "Not worker creator");
    require(block.number >= worker.deregisteredAt + lockPeriod
      (), "Worker is locked");

    uint256 bond = worker.bond;
    delete workers[workerId];

    tSQD.transfer(msg.sender, bond);

    emit WorkerWithdrawn(workerId, msg.sender);
  }
```

However, as long as the worker is not yet active (the current block has not yet reached the `registeredAt`), it can directly withdraw without calling `deregister` and waiting for the lock period. While this does not impact the TVL calculation, it violates the designed withdrawal flow and does not remove the worker from the `activeWorkerIds` array which will enable the unbounded loop attack vector.

Added POC to `WorkerRegistration.withdraw.t.sol`.

```
function testImmediatelyWithdraw() public {
    vm.roll(176329477);
    workerRegistration.register(workerId);
    workerRegistration.withdraw(workerId);
  }
```

Recommendations

Consider to add extra check when `withdraw` is performed :

```
function withdraw(bytes calldata peerId) external whenNotPaused {
    uint256 workerId = workerIds[peerId];
    require(workerId != 0, "Worker not registered");
    Worker storage worker = workers[workerId];
    require(!isWorkerActive(worker), "Worker is active");
    require(worker.creator == msg.sender, "Not worker creator");
-    require(block.number >= worker.deregisteredAt + lockPeriod
- (), "Worker is locked");
+    require(block.number >= worker.deregisteredAt + lockPeriod
+ () && worker.deregisteredAt != 0, "Worker is locked");

    uint256 bond = worker.bond;
    delete workers[workerId];

    tSQD.transfer(msg.sender, bond);

    emit WorkerWithdrawn(workerId, msg.sender);
  }
```

[H-04] Lack of access control on tSQD's `registerTokenOnL2`

Severity

Impact: High, malicious attacker can set L2 custom address to different address to break the bridge token.

Likelihood: Medium, attacker can front-ran the `registerTokenOnL2` to break the bridge token.

Description

tSQD is designed so that it can be bridged from Ethereum (L1) to Arbitrum (L2) via Arbitrum's generic-custom gateway. However, the `registerTokenOnL2` function, which sets the L2 token address via `gateway.registerTokenToL2`, is not currently restricted.

```
function registerTokenOnL2(
    address l2CustomTokenAddress,
    uint256 maxSubmissionCostForCustomGateway,
    uint256 maxSubmissionCostForRouter,
    uint256 maxGasForCustomGateway,
    uint256 maxGasForRouter,
    uint256 gasPriceBid,
    uint256 valueForGateway,
    uint256 valueForRouter,
    address creditBackAddress
  ) public payable {
    require(!shouldRegisterGateway, "ALREADY_REGISTERED");
    shouldRegisterGateway = true;

    gateway.registerTokenToL2{value: valueForGateway}(

              l2CustomTokenAddress, maxGasForCustomGateway, gasPriceBid, maxSubmissionC
    );

    router.setGateway{value: valueForRouter}(
      address(
        gateway
      ), maxGasForRouter, gasPriceBid, maxSubmissionCostForRouter, creditBackAddress
    );

    shouldRegisterGateway = false;
  }
```

An attacker can front-run the `registerTokenOnL2` and put an incorrect address for `l2CustomTokenAddress` to break the bridge token. Once it is called, the L2 token cannot be changed inside the gateway.

Recommendations

Use the Ownable functionality inside tSQD and restrict `registerTokenOnL2` so that it can only be called by the owner/admin, as suggested by the Arbitrum bridge token design.

8.3. Medium Findings

[M-01] `setGatewayAddress` incorrectly updates address of the gateway

Severity

Impact: Medium, gateway will still has it's old address

Likelihood: Medium, only occurs if the user decides to set a new address for his gateway

Description

The gateway owner can set a new address for it using `setGatewayAddress` function. Unfortunately, this function only updates `gatewayByAddress` mapping, leaving `Gateway` struct intact

```
function setGatewayAddress(bytes calldata peerId, address newAddress) public {
    (Gateway storage gateway, bytes32 peerIdHash) = _getGateway(peerId);
    _requireOperator(gateway);

    if (gateway.ownAddress != address(0)) {
>>    delete gatewayByAddress[gateway.ownAddress];
    }

    if (address(newAddress) != address(0)) {
      require(gatewayByAddress[newAddress] == bytes32
        (0), "Gateway address already registered");
>>     gatewayByAddress[newAddress] = peerIdHash;
    }
```

If the user tries to call `allocateComputationUnits`, the contract will expect the old gateway address in `msg.sender`

```
function allocateComputationUnits
    (bytes calldata peerId, uint256[] calldata workerId, uint256[] calldata cus)
    external
    whenNotPaused
  {
    require(workerId.length == cus.length, "Length mismatch");
    (Gateway storage gateway,) = _getGateway(peerId);
>>  require(gateway.ownAddress == msg.sender, "Only gateway can allocate CUs");
```

Coded POC for `GatewayRegistry.unstake.t.sol`

```
function test_SetAddress() public {
    GatewayRegistry.Gateway memory gt = gatewayRegistry.getGateway(bytes
      ("peerId"));
    bytes32 peerIdHash = gatewayRegistry.gatewayByAddress(address(this));
    // check previous gateway address
    assertEq(gt.ownAddress, address(this));
    assertEq(peerIdHash, keccak256("peerId"));
    // try set a new one
    address newAddy = address(0x6a7e);
    gatewayRegistry.setGatewayAddress(bytes("peerId"), newAddy);
    gt = gatewayRegistry.getGateway(bytes("peerId"));
    peerIdHash = gatewayRegistry.gatewayByAddress(newAddy);
    assertEq(peerIdHash, keccak256("peerId"));
    // this will fail, since address is not updated
    assertEq(gt.ownAddress, newAddy);
  }
```

Recommendations

Update `ownAddress` variable as well

```
...
    if (address(newAddress) != address(0)) {
      require(gatewayByAddress[newAddress] == bytes32
        (0), "Gateway address already registered");
      gatewayByAddress[newAddress] = peerIdHash;
>>    gateway.ownAddress = newAddress;
    }
```

[M-02] `activeWorkerIds` have no size limit, can grow unbounded and gas-grief / cause OOG errors

Severity

Impact: High, Denial of service and high gas costs to users

Likelihood: Low, unprofitable due to bond submitted

Description

The function `register` in `WorkerRegistration.sol` contract adds a new element to the `activeWorkerIds` array whenever a new worker is registered. There are no checks for registered `peerId`s, so anyone can register any peerId

as a worker. The only restriction is that the user has to submit a bond amount of tokens in order to do so, which will get unlocked later.

The issue is that if this array grows too large, it will cause excessive gas costs for other users, since plenty functions loop over this array. An extreme scenario is when the gas cost to loop over the array exceeds the block gas limit, DOSing operations.

An example of such loops over `activeWorkerIds` is shown below.

```
function deregister(bytes calldata peerId) external whenNotPaused {
    for (uint256 i = 0; i < activeWorkerIds.length; i++) {
        if (activeWorkerIds[i] == workerId) {
            activeWorkerIds[i] = activeWorkerIds[
                activeWorkerIds.length - 1
            ];
            activeWorkerIds.pop();
            break;
        }
    }
```

```
function getActiveWorkers() public view returns (Worker[] memory) {
    for (uint256 i = 0; i < activeWorkerIds.length; i++) {
        uint256 workerId = activeWorkerIds[i];
        Worker storage worker = workers[workerId];
        if (isWorkerActive(worker)) {
            activeWorkers[activeIndex] = worker;
            activeIndex++;
        }
    }
```

This is also valid for the functions `getActiveWorkerIds` and `getActiveWorkerCount`.

Since a user can increase the gas costs of other users at no cost to themselves, this is an issue. The malicious user can always come back after the lock period and deregister and withdraw their bond amount. However, users who interacted with the protocol in between pays inflated gas amounts due to this manipulation.

An unlikely scenario is when the inflated array is too large to traverse in a single block due to the block gas limit and DOSes the entire protocol. However this will not be profitable by the attacker.

Recommendations

Use a whitelist of `peerIds`, and limit the size of the `activeWorkerIds` array to a reasonable amount.

[M-03] User can lock tokens from the TemporaryHolding for an "infinite" amount of time

Severity

Impact: Medium, admin won't be able to retrieve tokens after lock time has passed

Likelihood: Medium, attacker needs to be a temporary holding beneficiary

Description

`TemporaryHoldings.sol` allows the `beneficiary` address to use tSQD in the whitelisted protocol contracts for a limited amount of time

```
function execute(
    addressto,
    bytescalldatadata,
    uint256requiredApprove
  ) public returns (bytes memory
    require(_canExecute(msg.sender), "Not allowed to execute");
    require(router.networkController().isAllowedVestedTarget
      (to), "Target is not allowed");

    // It's not likely that following addresses will be allowed by network
    // controller, but just in case
    require(to != address(this), "Cannot call self");
    require(to != address(tSQD), "Cannot call tSQD");

    if (requiredApprove > 0) {
      tSQD.approve(to, requiredApprove);
    }
    return to.functionCall(data);
  }
```

after `lockedUntil` amount of time has passed `admin` regains control over the funds inside

```
function _canExecute(address executor) internal view override returns (bool) {
    if (block.timestamp < lockedUntil) {
      return executor == beneficiary;
    }
    return executor == admin;
  }
```

One of the whitelisted targets is the `GatewayRegistry`. A savvy beneficiary can stake tokens from `TemporaryHolding` for a time far in the future and enjoy boosted CU, while admin cannot unstake these tokens even after `lockedUntil`

```
function _stakeWithoutTransfer
    (bytes calldata peerId, uint256 amount, uint128 durationBlocks) internal {
    (Gateway storage gateway, bytes32 peerIdHash) = _getGateway(peerId);
    _requireOperator(gateway);

    uint256 _computationUnits = computationUnitsAmount(amount, durationBlocks);
    uint128 lockStart = router.networkController().nextEpoch();
>>  uint128 lockEnd = lockStart + durationBlocks;
    ...
}
```

Recommendations

The easiest solution would be to disallow using gateway registry through `TemporaryHolding.sol`. If it's necessary for us to keep it as a valid target, consider decoding the payload in the `execute` function, and if it is the `stake` or `extend` functions, validate the duration.

[M-04] Protocol uses manipulatable randomness

Severity

Impact: Medium, distributors are generally trusted actors, but this could be a vector for manipulation.

Likelihood: Medium, randomness can be manipulated via multiple vectors

Description

The protocol generates a random number to choose which distributor will be able to commit rewards for a span of blocks. The issue is in the way the protocol generates this randomness.

```
function distributorIndex() public view returns (uint256) {
    uint256 slotStart = (block.number / 256) * 256;
    return uint256(blockhash(slotStart)) % distributors.length();
}
```

After this `distributionIndex` is calculated, it is used to select a distributor from the array.

There are two issues with this design:

1. Using `blockhash` is an unsafe way to generate randomness
2. The `slotStart` is manipulatable every 256 blocks

1. `blockhash` as source of randomness

In Arbitrum, transactions are ordered in first-come-first-serve ordering, and the sequencer responsible for ordering the transactions is centralized. However, the Arbitrum DAO plans to decentralize the sequencer in the future, meaning that malicious sequencers can keep including and broadcasting transactions until they can get a favourable hash for the block. Due to the nature of Arbitrum this is more difficult to pull of than in Ethereum mainnet, but is still possible once sequencer decentralization is achieved.

2. `slotStart` is manipulatable

`slotStart` is calculated as `(block.number / 256) * 256`. Now every time the `block.number` is a multiple of 256, `slotStart` will be calculated as the `block.number` itself.

According to the ethereum yellow paper, `blockhash` of the current block always returns 0. This is because the block hash of the current block hasnt been calculated yet. So every time the `block.number` is a multiple of 256, the `uint256(blockhash(slotStart))` will be calculated as 0. This allows the 0th distributor to make a commit everytime the `block.number` is a multiple of 256.

Since the randomness of the system is broken in these two ways, this is an issue

Recommendations

Firstly, `blockhash` should never be called on the current block, which is what happens in scenario 2 above. So using the bloackhash of `slotStart-1` will give the same effect without the 0th index being able to manipulate the randomness.

Secondly, in the future the blockhash on arbitrum might be more manipulatable. considering using chainlink VRF to generate randomness in a

more robust manner.

[M-05] Vesting contract can lead to frequent reverts

Severity

Impact: Medium, temporary DOS of funds

Likelihood: Medium, occurs when funds are staked / used in the protocol

Description

The `Vesting` is used to lock funds on behalf of a user while giving them the ability to participate in the protocol. The idea behind the contract is that the amount of funds in the contract will gradually be unlocked over time, while giving the owner the ability to use those tokens to stake or register workers and gateways with. This allows the users to use the tokens in the ecosystem, without having the option to withdraw them all at once.

The `VestingWallet` OpenZeppelin contract tracks a `_erc20Released` variable which keeps track of the tokens already paid out. When trigerring a new payout, the amount of tokens available is calculated as shown below.

```
function releasable() public view virtual returns (uint256) {
    return vestedAmount(uint64(block.timestamp)) - released();
}
```

The issue is that since the contract uses the `erc20.balanceOf` function to track the vesting amounts, this above expression can underflow and revert. This is because the balance in the contract can decrease if the user wishes to allocate some of the vested amount to staking or registering workers and gateways.

This is best demonstrated in the POC below.

The issue is recreated in the following steps

1. The vesting contract is transferred in 8 eth of tokens.
2. At the midpoint, half (4 eth) tokens have been vested out. These tokens are collected by calling `release`. Thus `_erc20Released` is set to 4 eth for the token.

3. Of the remaining 4 eth in the contract, 2 eth is allocated to staking.
4. After some time, more tokens are vested out.
5. Now when `release` is called, the function reverts. This is because `vestedAmount()` returns a value less than 4 eth, and `_erc20Released` is 4 eth. This causes the `releasable` function to underflow and revert.

So even though the contract has funds and can afford to pay out some vested rewards, this function reverts.

```
function test_AttackVesting() public {
    token.transfer(address(vesting), 8 ether);

    // Half (4 eth) is vested out
    vm.warp(vesting.start() + vesting.duration() / 2);
    vesting.release();
    assert(vesting.released(address(token)) == 4 ether);

    // Stake half of rest (2 ETH)
    bytes memory call = abi.encodeWithSelector(
        Staking.deposit.selector,
        0,
        2 ether
    );
    vesting.execute(address(router.staking()), call, 2 ether);

    // pass some time
    vm.warp(vesting.start() + (vesting.duration() * 60) / 100);
    // check rewards
    vm.expectRevert();
    vesting.release();
}
```

This causes a temporary DOS, and users are unable to release vested tokens until their stake or registration is paid out.

Since users lose access to part of the funds they deserve, this is a medium severity issue.

Recommendations

Consider adding a `depositForVesting(unit amount)` function, and tracking the amount with a storage variable `baseAmount` updated in this function. This way, the vesting rewards will calculate rewards based on this and not the `erc20.balanceOf` value. The result is that we need not decrease the `baseAmount` when tokens are sent out for staking, and then the vested amounts will be correctly calculated based on the value of the contract, instead of just the balances. This will prevent scenarios where claiming can revert even if funds are available.

8.4. Low Findings

[L-01] `PUSH0` not supported on Arbitrum

The contracts use solidity version `0.8.20`, which introduces the `PUSH0` opcode. This opcode is not present on Arbitrum. The `foundry.toml` file doe not specify any `evm_version`, so the contracts are compiled with the default `paris` version, which does not generate `PUSH0` opcodes, so the contracts are fine in the current state.

However if the `evm_version` is changed to `shanghai` or later, contracts will fail to deploy on Arbitrum due to the `PUSH0` opcodes.

It is recommended to either roll back the compiler version to `0.8.19` or earlier, or to specify the `evm_version=paris` in the `foundry.toml` file, to prevent any mistakes with the evm_version in the future.

[L-02] If the bond amount ever increased, there is no direct way to update workers bond

Currently, the `returnExcessiveBond` function is available inside the `WorkerRegistration`. In the event that the bond is ever reduced from the system, the excess amount can be claimed and returned to the creator. However, if the bond ever increases, there is no direct way for workers to update the bond, it has to go through the whole deregister and register process, which requires waiting for a lock period. This will add a time delay for the actual bond inside the system to match the TVL calculation. Consider to add function to update bond in case of the bond ever increased, similar to `returnExcessiveBond`.

[L-03] Worker is removed from the active workers list too early

During `deregister` call we remove worker from the `activeWorkerIds` array right away, however `deregisteredAt` value is set to the next epoch

```
function deregister(bytes calldata peerId) external whenNotPaused {
    uint256 workerId = workerIds[peerId];
    require(workerId != 0, "Worker not registered");
    require(isWorkerActive(workers[workerId]), "Worker not active");
    require(workers[workerId].creator == msg.sender, "Not worker creator");

>>    workers[workerId].deregisteredAt = nextEpoch();

    for (uint256 i = 0; i < activeWorkerIds.length; i++) {
      if (activeWorkerIds[i] == workerId) {
        activeWorkerIds[i] = activeWorkerIds[activeWorkerIds.length - 1];
        activeWorkerIds.pop();
        break;
      }
    }
```

and according to the `isWorkerActive`,

```
function isWorkerActive(Worker storage worker) internal view returns (bool) {
    return worker.registeredAt > 0 && worker.registeredAt <= block.number
      && (worker.deregisteredAt == 0 || worker.deregisteredAt >= block.number);
  }
```

worker is considered active if `deregisteredAt >= block.number`

[L-04] `deposit` function in Staking does not check whether the worker is still active

The staker will not receive any reward and must wait until the next epoch delay to withdraw the staked tokens if the provided worker is not currently active when calling the `deposit` function in the Staking contract. It is advisable to check if the provided worker is active in the `WorkerRegistration` contract when making a `deposit` call.

[L-05] Unsuccessful / overwritten commits cannot be approved again

The `DistributedRewardDistribution` allows distributors to commit a reward distribution and then have other distributors approve it. When the number of

approvals hit a threshold, the commit is considered approved and the rewards are distributed out.

During the commit process, the current commit is stored in a mapping as shown below:

```
commitments[fromBlock][toBlock] = commitment;
approves[fromBlock][toBlock] = 1;
```

The current distributor can always call this function, and if they use the same `fromBlock` and `toBlock`, they can always overwrite the previous commitment. This also resets the approves count to 1.

The issue is that in the `approve` function, a certain commit can only be approved once, no matter if its an overwritten or a new commit.

```
require(!alreadyApproved[commitment][msg.sender], "Already approved");
    approves[fromBlock][toBlock]++;
    alreadyApproved[commitment][msg.sender] = true;
```

Now consider the following scenario:

1. There are 10 distributors. The limit for approvals is 5.
2. Distributor 1 adds a commitment for blocks 1-100. 3 other distributors approve it.
3. After 256 blocks, distributor 2 gains control. They call `commit` again and overwrite the previous commitment with a different one, but for the same 1-100 blocks.
4. After another 256 blocks, distributor 3 gains control and resubmits the original commitment made by distributor 1 for 1-100 blocks. Now, the distributors who voted before cannot vote anymore, since `alreadyApproved[commitment][msg.sender]` is set to true. So even though the contract labels it as approved by the distributor, the `approves` mapping does not reflect that.

This can deny votes in case of resubmissions, and also breaks the invariant that the `approves` should contain the same number as the number of unique addresses whose `alreadyApproved` mapping is true.

Since we are resetting the `approves` count to 1, we should use a fresh `alreadyApproved` mapping. Consider adding a `commitmentId` variable which increases on every commit. The approvals mapping should then implement an

extra key and use `alreadyApproved[commitment][commitmentId][msg.sender]` instead, which will reset the already given approvals for the resubmitted commitment.