



Security Review For Seda Protocol



Public Best Efforts Audit Contest Prepared For: **Seda Protocol**

Lead Security Expert:

g

Date Audited:

February 17 - March 10, 2025

Introduction

Seda is programmable oracle infrastructure. Allowing developers to access any data from any L1 or L2.

This audit is focussed on the chain and the oracle batch proof accessibility and verification across evm chains.

Scope

Repository: [sedaprotocol/seda-chain](https://github.com/sedaprotocol/seda-chain)

Audited Commit: [58347a2cac169e697b84dd3d4d28669a76503cc3](https://github.com/sedaprotocol/seda-chain/commit/58347a2cac169e697b84dd3d4d28669a76503cc3)

Final Commit: [5a13ad044ad4806a3caca4a9d6ea1278619de68a](https://github.com/sedaprotocol/seda-chain/commit/5a13ad044ad4806a3caca4a9d6ea1278619de68a)

Files:

- [app/abci/errors.go](#)
- [app/abci/expected_keepers.go](#)
- [app/abci/handlers.go](#)
- [app/ante.go](#)
- [app/app.go](#)
- [app/encoding.go](#)
- [app/export.go](#)
- [app/genesis.go](#)
- [app/keepers/keepers.go](#)
- [app/modules.go](#)
- [app/params/config.go](#)
- [app/params/encoding.go](#)
- [app/upgrades.go](#)
- [app/upgrades/mainnet/v1/constants.go](#)
- [app/upgrades/types.go](#)
- [app/utils/merkle.go](#)
- [app/utils/merkle_proof.go](#)
- [app/utils/print_info.go](#)
- [app/utils/seda_keys.go](#)
- [app/utils/seda_signer.go](#)
- [app/wasm.go](#)

- cmd/sedad/cmd/config_defaults.go
- cmd/sedad/cmd/genaccounts.go
- cmd/sedad/cmd/git_download.go
- cmd/sedad/cmd/init.go
- cmd/sedad/cmd/init_cmds.go
- cmd/sedad/cmd/root.go
- cmd/sedad/cmd/rosetta_disabled.go
- cmd/sedad/cmd/rosetta_enabled.go
- cmd/sedad/gentx/collect_gentxs.go
- cmd/sedad/gentx/gentx.go
- cmd/sedad/main.go
- go.mod
- go.sum
- proto/sedachain/batching/v1/batching.proto
- proto/sedachain/batching/v1/evidence.proto
- proto/sedachain/batching/v1/genesis.proto
- proto/sedachain/batching/v1/query.proto
- proto/sedachain/data_proxy/v1/data_proxy.proto
- proto/sedachain/data_proxy/v1/genesis.proto
- proto/sedachain/data_proxy/v1/query.proto
- proto/sedachain/data_proxy/v1/tx.proto
- proto/sedachain/pubkey/v1/genesis.proto
- proto/sedachain/pubkey/v1/pubkey.proto
- proto/sedachain/pubkey/v1/query.proto
- proto/sedachain/pubkey/v1/tx.proto
- proto/sedachain/staking/v1/tx.proto
- proto/sedachain/tally/v1/genesis.proto
- proto/sedachain/tally/v1/query.proto
- proto/sedachain/tally/v1/tally.proto
- proto/sedachain/tally/v1/tx.proto
- proto/sedachain/vesting/v1/tx.proto

- proto/sedachain/vesting/v1/vesting.proto
- proto/sedachain/wasm_storage/v1/genesis.proto
- proto/sedachain/wasm_storage/v1/query.proto
- proto/sedachain/wasm_storage/v1/tx.proto
- proto/sedachain/wasm_storage/v1/wasm_storage.proto
- tools/tools.go
- x/batching/client/cli/query.go
- x/batching/client/cli/tx.go
- x/batching/keeper/batch.go
- x/batching/keeper/data_result.go
- x/batching/keeper/endblock.go
- x/batching/keeper/evidence.go
- x/batching/keeper/genesis.go
- x/batching/keeper/keeper.go
- x/batching/keeper/querier.go
- x/batching/module.go
- x/batching/types/batch.go
- x/batching/types/codec.go
- x/batching/types/data_result.go
- x/batching/types/errors.go
- x/batching/types/events.go
- x/batching/types/evidence.go
- x/batching/types/expected_keepers.go
- x/batching/types/genesis.go
- x/batching/types/keys.go
- x/data-proxy/client/cli/query.go
- x/data-proxy/client/cli/tx.go
- x/data-proxy/keeper/abci.go
- x/data-proxy/keeper/genesis.go
- x/data-proxy/keeper/grpc_query.go
- x/data-proxy/keeper/keeper.go

- x/data-proxy/keeper/msg_server.go
- x/data-proxy/keeper/params.go
- x/data-proxy/module.go
- x/data-proxy/types/codec.go
- x/data-proxy/types/errors.go
- x/data-proxy/types/events.go
- x/data-proxy/types/genesis.go
- x/data-proxy/types/keys.go
- x/data-proxy/types/params.go
- x/data-proxy/types/proxy_config.go
- x/data-proxy/types/tx.go
- x/pubkey/client/cli/keyfile.go
- x/pubkey/client/cli/query.go
- x/pubkey/client/cli/tx.go
- x/pubkey/keeper/endblock.go
- x/pubkey/keeper/genesis.go
- x/pubkey/keeper/grpc_query.go
- x/pubkey/keeper/keeper.go
- x/pubkey/keeper/msg_server.go
- x/pubkey/module.go
- x/pubkey/types/codec.go
- x/pubkey/types/events.go
- x/pubkey/types/expected_keepers.go
- x/pubkey/types/genesis.go
- x/pubkey/types/keys.go
- x/pubkey/types/params.go
- x/pubkey/types/tx.go
- x/slashing/module.go
- x/slashing/msg_server.go
- x/staking/autocli.go
- x/staking/client/cli/tx.go

- x/staking/client/cli/utils.go
- x/staking/keeper/invariants.go
- x/staking/keeper/keeper.go
- x/staking/keeper/msg_server.go
- x/staking/module.go
- x/staking/types/codec.go
- x/staking/types/expected_keepers.go
- x/staking/types/msg.go
- x/tally/client/cli/query.go
- x/tally/keeper/endblock.go
- x/tally/keeper/filter.go
- x/tally/keeper/gas_meter.go
- x/tally/keeper/genesis.go
- x/tally/keeper/grpc_query.go
- x/tally/keeper/keeper.go
- x/tally/keeper/msg_server.go
- x/tally/keeper/tally_vm.go
- x/tally/module.go
- x/tally/types/abci_types.go
- x/tally/types/codec.go
- x/tally/types/data_result.go
- x/tally/types/errors.go
- x/tally/types/events.go
- x/tally/types/expected_keepers.go
- x/tally/types/filters.go
- x/tally/types/filters_util.go
- x/tally/types/gas_meter.go
- x/tally/types/genesis.go
- x/tally/types/keys.go
- x/tally/types/params.go
- x/tally/types/sigma_multiplier.go

- x/tally/types/telemetry.go
- x/tally/types/types.go
- x/tally/types/vm_exit_codes.go
- x/vesting/client/cli/tx.go
- x/vesting/keeper/msg_server.go
- x/vesting/module.go
- x/vesting/types/codec.go
- x/vesting/types/errors.go
- x/vesting/types/expected_keepers.go
- x/vesting/types/messages.go
- x/vesting/types/types.go
- x/vesting/types/vesting.go
- x/wasm-storage/client/cli/decoder.go
- x/wasm-storage/client/cli/gov_tx.go
- x/wasm-storage/client/cli/query.go
- x/wasm-storage/client/cli/tx.go
- x/wasm-storage/keeper/genesis.go
- x/wasm-storage/keeper/keeper.go
- x/wasm-storage/keeper/msg_server.go
- x/wasm-storage/keeper/querier.go
- x/wasm-storage/module.go
- x/wasm-storage/types/codec.go
- x/wasm-storage/types/errors.go
- x/wasm-storage/types/events.go
- x/wasm-storage/types/expected_keepers.go
- x/wasm-storage/types/genesis.go
- x/wasm-storage/types/keys.go
- x/wasm-storage/types/messages.go
- x/wasm-storage/types/params.go
- x/wasm-storage/types/wasm.go

Repository: `sedaprotocol/seda-chain-contracts`

Audited Commit: `9ab67eda2bd67e728f1bbf8379c9581431dee21d`

Final Commit: `cda47a6420cealb9e7a3cfcd3845888559a8c3b7`

Files:

- `contract/src/consts.rs`
- `contract/src/contract.rs`
- `contract/src/error.rs`
- `contract/src/messages/data_requests/execute/commit_result.rs`
- `contract/src/messages/data_requests/execute/dr_events.rs`
- `contract/src/messages/data_requests/execute/mod.rs`
- `contract/src/messages/data_requests/execute/post_request.rs`
- `contract/src/messages/data_requests/execute/reveal_result.rs`
- `contract/src/messages/data_requests/execute/set_timeout_config.rs`
- `contract/src/messages/data_requests/mod.rs`
- `contract/src/messages/data_requests/query.rs`
- `contract/src/messages/data_requests/state/data_requests_map.rs`
- `contract/src/messages/data_requests/state/mod.rs`
- `contract/src/messages/data_requests/state/timeouts.rs`
- `contract/src/messages/data_requests/sudo/expire_data_requests.rs`
- `contract/src/messages/data_requests/sudo/mod.rs`
- `contract/src/messages/data_requests/sudo/remove_requests.rs`
- `contract/src/messages/enumerable_set.rs`
- `contract/src/messages/mod.rs`
- `contract/src/messages/owner/execute/accept_ownership.rs`
- `contract/src/messages/owner/execute/add_to_allowlist.rs`
- `contract/src/messages/owner/execute/mod.rs`
- `contract/src/messages/owner/execute/pause.rs`
- `contract/src/messages/owner/execute/remove_from_allowlist.rs`
- `contract/src/messages/owner/execute/transfer_ownership.rs`
- `contract/src/messages/owner/execute/unpause.rs`
- `contract/src/messages/owner/mod.rs`

- `contract/src/messages/owner/query.rs`
- `contract/src/messages/owner/state.rs`
- `contract/src/messages/owner/utils.rs`
- `contract/src/messages/staking/execute/mod.rs`
- `contract/src/messages/staking/execute/set_staking_config.rs`
- `contract/src/messages/staking/execute/stake.rs`
- `contract/src/messages/staking/execute/staking_events.rs`
- `contract/src/messages/staking/execute/unstake.rs`
- `contract/src/messages/staking/execute/withdraw.rs`
- `contract/src/messages/staking/mod.rs`
- `contract/src/messages/staking/query.rs`
- `contract/src/messages/staking/state/is_eligible_for_dr.rs`
- `contract/src/messages/staking/state/mod.rs`
- `contract/src/messages/staking/state/stakers_map.rs`
- `contract/src/state.rs`
- `contract/src/types.rs`
- `contract/src/utils.rs`

Repository: [sedaprotocol/seda-evm-contracts](https://github.com/sedaprotocol/seda-evm-contracts)

Audited Commit: `07f329125cb62eacce32f5057ba8e59fc4c67f7d`

Final Commit: `2eac4bb4d3ea2d0cbf967950f8ed48feeb7e0ec3`

Files:

- `contracts/core/SedaCoreV1.sol`
- `contracts/core/abstract/RequestHandlerBase.sol`
- `contracts/core/abstract/ResultHandlerBase.sol`
- `contracts/interfaces/IProver.sol`
- `contracts/interfaces/IRequestHandler.sol`
- `contracts/interfaces/IResultHandler.sol`
- `contracts/interfaces/ISedaCore.sol`
- `contracts/libraries/SedaDataTypes.sol`
- `contracts/provers/Secp256k1ProverV1.sol`

- `contracts/provers/abstract/ProverBase.sol`
-

Repository: `sedaprotocol/seda-wasm-vm`

Audited Commit: `8a0a7274119150b72c02cedce3edcc16b43d1599`

Final Commit: `a45d55757c78015d4e8b98cdfb7125b1c615ed99`

Files:

- `libtallyvm/build.rs`
- `libtallyvm/cbindgen.toml`
- `libtallyvm/src/errors.rs`
- `libtallyvm/src/lib.rs`
- `runtime/core/src/context.rs`
- `runtime/core/src/core_vm_imports/call_result.rs`
- `runtime/core/src/core_vm_imports/execution_result.rs`
- `runtime/core/src/core_vm_imports/keccak256.rs`
- `runtime/core/src/core_vm_imports/mod.rs`
- `runtime/core/src/core_vm_imports/secp256_k1.rs`
- `runtime/core/src/errors.rs`
- `runtime/core/src/lib.rs`
- `runtime/core/src/metering.rs`
- `runtime/core/src/resources_dir.rs`
- `runtime/core/src/runtime.rs`
- `runtime/core/src/runtime_context.rs`
- `runtime/core/src/safe_wasi_imports.rs`
- `runtime/core/src/tally_vm_imports/http_fetch.rs`
- `runtime/core/src/tally_vm_imports/mod.rs`
- `runtime/core/src/tally_vm_imports/proxy_http_fetch.rs`
- `runtime/core/src/vm_imports.rs`
- `runtime/core/src/wasm_cache.rs`
- `tallyvm/execute.go`
- `tallyvm/lib seda_tally_vm.h`
- `tallyvm/link_glibclinux_aarch64.go`

- [tallyvm/link_glibclinux_x86_64.go](#)
- [tallyvm/link_mac.go](#)
- [tallyvm/link_muslc_aarch64.go](#)
- [tallyvm/link_muslc_x86_64.go](#)

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

High	Medium
19	15

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

000000	Kodyvim	gegul
0x0x0xw3	PASCAL	gxh191
0xHammad	PNS	leopoldflint
0xMgwan	Schnilch	newspacexyz
0xNirix	Stiglitz	oxelmiguel
0xeix	abdulsamijay	rsam_eth
0xlookman	blutorque	stonejiajia
4n0nx	bronze_pickaxe	stuart_the_minion
Boy2000	cu5t0mPe0	tallo
ChaosSR	destiny_rs	verbotenviking
DeLaSoul	dod4ufn	x0lohacllohell
HarryBarz	g	zxripor

Issue H-1: Chain halt as VerifyVoteExtensionHandler is not guaranteed to run

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/30>

Found by

Boy2000, stuart_the_minion

Summary

Late/slow precommit votes, after +2/3 votes do not trigger VerifyVoteExtensionHandler. ProcessProposalHandler expects all vote extensions to be valid. Single malicious/misbehaving validator can inject invalid vote extension to the (previous) block, resulting in chain halt.

Root Cause

1. PrepareProposal does not re-verify/ignore invalid LocalLastCommit votes

<https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-chain/app/abci/handlers.go#L177>

```
func (h *Handlers) PrepareProposalHandler() sdk.PrepareProposalHandler {
    return func(ctx sdk.Context, req *abcitypes.RequestPrepareProposal)
    ↪ (*abcitypes.ResponsePrepareProposal, error) {
        // Check if there is a batch whose signatures must be collected
        // at this block height.
        var collectSigs bool
        _, err := h.batchingKeeper.GetBatchForHeight(ctx,
    ↪ ctx.BlockHeight()+BlockOffsetCollectPhase)
        if err != nil {
            if !errors.Is(err, collections.ErrNotFound) {
                return nil, err
            }
        } else {
            collectSigs = true
        }

        var injection []byte
        if req.Height > ctx.ConsensusParams().Abci.VoteExtensionsEnableHeight &&
    ↪ collectSigs {
            err := baseapp.ValidateVoteExtensions(ctx, h.stakingKeeper, req.Height,
    ↪ ctx.ChainID(), req.LocalLastCommit)
            if err != nil {
                return nil, err
            }
        }
    }
```

```

    }

@>    injection, err = json.Marshal(req.LocalLastCommit)
    if err != nil {
        h.logger.Error("failed to marshal extended votes", "err", err)
        return nil, err
    }

    injectionSize := int64(len(injection))
    if injectionSize > req.MaxTxBytes {
        h.logger.Error(
            "vote extension size exceeds block size limit",
            "injection_size", injectionSize,
            "MaxTxBytes", req.MaxTxBytes,
        )
        return nil, ErrVoteExtensionInjectionTooBig
    }
    req.MaxTxBytes -= injectionSize
}

defaultRes, err := h.defaultPrepareProposal(ctx, req)
if err != nil {
    h.logger.Error("failed to run default prepare proposal handler", "err",
↪ err)
    return nil, err
}

proposalTxs := defaultRes.Txs
if injection != nil {
    proposalTxs = append([][]byte{injection}, proposalTxs...)
    h.logger.Debug("injected local last commit", "height", req.Height)
}
return &abcitypes.ResponsePrepareProposal{
    Txs: proposalTxs,
}, nil
}
}

```

2. ProcessProposalHandler expects all votes (batches) to be valid/verified

<https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-chain/app/abci/handlers.go#L235>

```

func (h *Handlers) ProcessProposalHandler() sdk.ProcessProposalHandler {
    return func(ctx sdk.Context, req *abcitypes.RequestProcessProposal)
↪ (*abcitypes.ResponseProcessProposal, error) {
        if req.Height <= ctx.ConsensusParams().Abci.VoteExtensionsEnableHeight {
            return h.defaultProcessProposal(ctx, req)
        }
    }
}

```

```

        batch, err := h.batchingKeeper.GetBatchForHeight(ctx,
↪ ctx.BlockHeight()+BlockOffsetCollectPhase)
        if err != nil {
            if errors.Is(err, collections.ErrNotFound) {
                return h.defaultProcessProposal(ctx, req)
            }
            return nil, err
        }

        var extendedVotes abcitypes.ExtendedCommitInfo
        if err := json.Unmarshal(req.Txs[0], &extendedVotes); err != nil {
            h.logger.Error("failed to decode injected extended votes tx", "err",
↪ err)
            return nil, err
        }

        // Validate vote extensions and batch signatures.
        err = baseapp.ValidateVoteExtensions(ctx, h.stakingKeeper, req.Height,
↪ ctx.ChainID(), extendedVotes)
        if err != nil {
            return nil, err
        }

        for _, vote := range extendedVotes.Votes {
            // Only consider extensions with pre-commit votes.
            if vote.BlockIdFlag == cmttypes.BlockIdFlagCommit {
                err = h.verifyBatchSignatures(ctx, batch.BatchNumber,
↪ batch.BatchId, vote.VoteExtension, vote.Validator.Address)
                if err != nil {
@> h.logger.Error("proposal contains an invalid vote extension",
↪ "vote", vote)
                return nil, err
            }
        }
    }

    req.Txs = req.Txs[1:]
    return h.defaultProcessProposal(ctx, req)
}

```

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

1. Malicious/misbehaving validator late-injects invalid vote at the right height (when there is a batch to sign, to not return early due to 0 batches). Alternatively optimistically sends invalid votes continuously.
2. The vote is not validated (with app logic) and is added to LocalLastCommit.
3. Next block proposer -> propose block with invalid vote -> ProcessProposalHandler repeatedly fails.

Impact

Chain halt

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/531>

gjaldon

This issue was fixed by verifying the batch signatures of every vote extension in `ProcessProposal()`. Every invalid vote extension will be pruned. This addresses any invalid vote extensions that sneaked in post-quorum.

Issue H-2: Malicious validators will bypass consensus threshold requirements affecting the integrity of the SEDA protocol's cross-chain data verification system

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/62>

Found by

0x0x0xw3, 0xlookman, 4n0nx, Boy2000, ChaosSR, Kodyvim, PASCAL, Schnilch, Stiglitz, abdulsamijay, destiny_rs, dod4ufn, g, newspacexyz, oxelmiguel, rsam_eth, tallo, verbotenviking, zxripter

Summary

A lack of uniqueness validation in validator signature processing will cause a critical security breach for the SEDA protocol as malicious validators will artificially inflate their voting power by submitting duplicate signatures, allowing validators with minimal actual authority to unilaterally approve batches.

Root Cause

In [seda-evm-contracts/contracts/provers/Secp256k1ProverV1.sol:114-119](#) the contract accumulates validator voting power without checking if a validator's signature has already been counted:

```
for (uint256 i = 0; i < validatorProofs.length; i++) {
    if (!_verifyValidatorProof(validatorProofs[i], s.lastValidatorsRoot)) {
        revert InvalidValidatorProof();
    }
    if (!_verifySignature(batchId, signatures[i], validatorProofs[i].signer)) {
        revert InvalidSignature();
    }
    votingPower += validatorProofs[i].votingPower;
}
```

The code fails to track which validators have already contributed to the voting power total, allowing duplicate entries.

Internal Pre-conditions

1. A malicious validator needs to control at least one validator with any amount of voting power
2. The validator must have the ability to submit transactions to the `Secp256k1ProverV1` contract with multiple copies of their own signature and proof

External Pre-conditions

None required - the attack can be executed independently of external systems

Attack Path

1. Attacker identifies their validator's voting power (e.g., 0.25% of total power)
2. Attacker calculates how many duplicates are needed to reach the 66.67% consensus threshold
3. Attacker generates a valid signature for their validator for a malicious batch
4. Attacker creates arrays containing their signature and validator proof repeated the required number of times
5. Attacker calls `postBatch()` with these arrays, passing the verification checks
6. The contract approves the batch despite only having one actual validator's approval

Impact

The SEDA protocol suffers a complete security breakdown. Validators with minimal voting power can unilaterally approve batches, completely bypassing the consensus mechanism. This allows malicious validators to:

- Post fraudulent data results
- Update the validator set with malicious entries
- Potentially steal funds from dependent contracts that trust SEDA's data
- Undermine the foundational security assumptions of the cross-chain oracle system

PoC

```
// Add test case at: `seda-evm-contracts/test/prover/Secp256k1ProverV1.test.ts`
it('proves vulnerability scales with validator power percentage', async () => {
  // Create test fixtures with different validator power distributions
  const distributions = [
```

```

    { description: "Tiny validator (0.25%)", validatorCount: 100, validatorIndex: 1
    ↪ },
    { description: "Small validator (1%)", validatorCount: 25, validatorIndex: 1 },
    { description: "Medium validator (5%)", validatorCount: 10, validatorIndex: 2 },
    { description: "Larger validator (10%)", validatorCount: 9, validatorIndex: 1 }
  ];

  for (const dist of distributions) {
    console.log(`\nTesting: ${dist.description}`);

    // Deploy with specific validator count
    const { prover: customProver, data: customData } = await deployWithSize({
    ↪ validators: dist.validatorCount });
    const customWallets = customData.wallets;

    // Get validator power and calculate percentage
    const validatorPower =
    ↪ customData.validatorProofs[dist.validatorIndex].votingPower;
    const totalPower = 100000000; // Default total
    const powerPercentage = (validatorPower / totalPower) * 100;
    console.log(`Validator has ${validatorPower} voting power
    ↪ (${powerPercentage.toFixed(2)}% of total)`);

    // Create batch and sign with this validator
    const { newBatchId, newBatch } =
    ↪ generateNewBatchWithId(customData.initialBatch);
    const signature = await
    ↪ customWallets[dist.validatorIndex].signingKey.sign(newBatchId).serialized;

    // Verify single signature doesn't reach consensus
    try {
      await customProver.postBatch(newBatch, [signature],
    ↪ [customData.validatorProofs[dist.validatorIndex]]);
      console.log("ERROR: Single signature was enough - test invalid");
    } catch (e) {
      // Calculate required duplicates
      const consensusThreshold = 66_670_000; // 66.67%
      const duplicatesNeeded = Math.ceil(consensusThreshold / validatorPower);
      console.log(`Need ${duplicatesNeeded} duplicates to reach consensus
    ↪ threshold`);

      // Prepare duplicated arrays
      const signatures = Array(duplicatesNeeded).fill(signature);
      const duplicatedProofs =
    ↪ Array(duplicatesNeeded).fill(customData.validatorProofs[dist.validatorIndex]);

      const [batchSender] = await ethers.getSigners();

      // Demonstrate exploit
      await expect(customProver.postBatch(newBatch, signatures, duplicatedProofs))

```

```

        .to.emit(customProver, 'BatchPosted')
        .withArgs(newBatch.batchHeight, newBatchId, batchSender.address);

    // Verify success
    const lastBatchHeight = await customProver.getLastBatchHeight();
    expect(lastBatchHeight).to.equal(newBatch.batchHeight);

    console.log(`EXPLOIT SUCCESSFUL: ${powerPercentage.toFixed(2)}% validator can
↪ update batches with ${duplicatesNeeded} duplicates`);
    }
  }
});

```

The results demonstrate how validators with any amount of power can exploit the vulnerability:

Validator Power	Duplicates Required	Notes
0.25%	267	Even tiny validators can exploit the vulnerability
1%	67	Small validators need moderate duplication
5%	14	Medium validators need minimal duplication
10%	7	Larger validators need very few duplicates

The lower the validator's power, the more duplicates are needed, but all validators can eventually reach the threshold.

Command to run test:

```

npx hardhat test test/prover/Secp256k1ProverV1.test.ts --grep "proves vulnerability
↪ scales with validator power percentage"

```

Result example:

```

Secp256k1ProverV1

Testing: Tiny validator (0.25%)
Validator has 252525 voting power (0.25% of total)
Need 265 duplicates to reach consensus threshold
EXPLOIT SUCCESSFUL: 0.25% validator can update batches with 265 duplicates

Testing: Small validator (1%)
Validator has 1041666 voting power (1.04% of total)
Need 65 duplicates to reach consensus threshold
EXPLOIT SUCCESSFUL: 1.04% validator can update batches with 65 duplicates

```

```
Testing: Medium validator (5%)
Validator has 2777777 voting power (2.78% of total)
Need 25 duplicates to reach consensus threshold
EXPLOIT SUCCESSFUL: 2.78% validator can update batches with 25 duplicates

Testing: Larger validator (10%)
Validator has 3125000 voting power (3.13% of total)
Need 22 duplicates to reach consensus threshold
EXPLOIT SUCCESSFUL: 3.13% validator can update batches with 22 duplicates
    proves vulnerability scales with validator power percentage (1559ms)

1 passing (2s)
```

Mitigation

Add uniqueness tracking to prevent counting the same validator more than once:

```
function postBatch(
    SedaDataTypes.Batch calldata newBatch,
    bytes[] calldata signatures,
    SedaDataTypes.ValidatorProof[] calldata validatorProofs
) external override(ProverBase) whenNotPaused {
    // ... existing code ...

    uint64 votingPower = 0;
    mapping(address => bool) memory seenValidators;

    for (uint256 i = 0; i < validatorProofs.length; i++) {
        address signer = validatorProofs[i].signer;

        // Prevent duplicate validators
        if (seenValidators[signer]) {
            revert DuplicateValidator(signer);
        }
        seenValidators[signer] = true;

        // ... existing verification code ...

        votingPower += validatorProofs[i].votingPower;
    }

    // ... rest of function ...
}
```

This fix ensures each validator is only counted once when calculating the total voting power, preserving the integrity of the consensus mechanism.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-evm-contracts/pull/95>

gjaldon

The strictly increasing order of signers when posting a batch addresses this issue.

Issue H-3: ExecuteTallyVM has a memory leak which will lead to nodes eventually crashing

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/182>

Found by

Boy2000, g, gxh191, tallo, zxripter

Summary

The continuous memory allocation without freeing in `execute.go` will cause a memory leak for node operators as each block will lose a small amount of memory, eventually depleting the node's resources over extended operation periods.

Root Cause

In `execute.go` the `configDirC` C string is allocated with `C.CString(LogDir)` but never freed with `C.free(unsafe.Pointer(configDirC))`, unlike all other C string allocations in the same function.

<https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-wasm-vm/tallyvm/execute.go#L27>

Internal Pre-conditions

1. The SEDA node is running and processing blocks
2. The `ExecuteTallyVm` function is called from the `EndBlock` function
3. The `LogDir` global variable is set to a non-empty string path

External Pre-conditions

None

Attack Path

1. The Tally module's `EndBlock` function is called automatically at the end of each block
2. This calls `ProcessTallies` which processes data requests ready for tallying
3. For each tally operation, the `FilterAndTally` method is invoked

4. This ultimately executes the `ExecuteTallyVm` function in `execute.go`
5. Each execution allocates memory for `configDirC` without freeing it
6. The allocated memory accumulates over time as each block is processed
7. Eventually, after running for an extended period, the node will run out of memory

Impact

The SEDA node operators suffer a continuous memory leak that will eventually lead to degraded performance and potential node crashes. The amount of memory leaked per block is equal to the length of the `LogDir` string plus 1 byte (for the null terminator). For a typical path length of ~30 bytes, `/home/seda/.seda/tally-logs\00`, a default `MaxTalliesPerBlock` of 100, and an average of 20 data requests per block, this will add up to ~3gb of allocated memory that is leaked each year. Longer paths, more data requests per block will result in a faster leakage and result in an eventual crash for all systems. Systems with lower specs will crash sooner rather than later

Mitigation

just like with all the other `C.Cstring` initializations in the function, add the line:

```
defer C.free(unsafe.Pointer(configDirC))
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-wasm-vm/pull/60>

gjaldon

The fix applies the recommended mitigation of freeing the `configDirC` `Cstring`.

Issue H-4: Gas costs are severely underpriced for certain WASM instructions which can lead to network DoS

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/183>

Found by

tallo

Summary

Insufficient gas pricing for complex WebAssembly operations will cause a denial of service vulnerability for the SEDA Protocol as malicious actors will exploit underpriced operations like TableGrow, MemoryFill, and complex math functions to consume disproportionate computational resources while paying minimal gas fees.

Root Cause

metering.rs uses a flat cost model for all instructions except for accounting operations and MemoryGrow. <https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-wasm-vm/runtime/core/src/metering.rs#L51>

```
pub fn get_wasm_operation_gas_cost(operator: &Operator) -> u64 {
    if is_accounting(operator) {
        return GAS_PER_OPERATION * GAS_ACCOUNTING_MULTIPLIER;
    }

    match operator {
        Operator::MemoryGrow { mem, mem_byte: _ } => {
            GAS_MEMORY_GROW_BASE + ((WASM_PAGE_SIZE as u64 * *mem as u64) *
↳ GAS_PER_BYTE)
        }
        _ => GAS_PER_OPERATION,
    }
}
```

The absence of specific, higher gas costs for computationally expensive operations like TableGrow, MemoryFill, TableCopy, and complex mathematical functions creates a significant imbalance between the gas paid and computational resources consumed.

Internal Pre-conditions

1. User oracle program needs to be registered with the blockchain
2. A data request needs to be made which makes use of the oracle program

External Pre-conditions

None

Attack Path

1. Attacker crafts a malicious Oracle Program that heavily utilizes underpriced operations (e.g., repeated TableGrow, MemoryFill, or computationally intensive mathematical functions)
2. Attacker submits this Oracle Program to the SEDA Protocol through a data request
3. Oracle Program executes on the SEDA Network, consuming disproportionate computational resources while paying minimal gas fees
4. Each execution consumes only GAS_PER_OPERATION (115) gas units despite potentially requiring orders of magnitude more computational resources
5. Multiple malicious requests can be submitted to amplify the attack's impact, causing significant resource exhaustion across the network

Impact

The impact changes depending on the instruction:

1. TableGrow: this instruction will allocate new table entries in WebAssembly, a malicious program could call this with a large size parameter and force the host environment to allocate a large amount of memory until the node crashes
2. MemoryFill/MemoryFill: These operations allows for filling a range of memory with a specific value. A malicious program could call this repeatedly call this to consume all computation resources and delay program execution termination indefinitely
3. MemoryInit/TableInit: similar to MemoryFill
4. Sin/cos/exp/log/div/sqrt: much more expensive than the more basic operations and should have a higher gas price

Considering that all these operations are done in the context of Endblock, each one of these can severely DoS the entire network and cause nodes to crash. At a price of 115 gas for each instruction, its essentially free for a malicious program to severely harm the network.

Tally WASM VM Robustness: The robustness of the Tally WASM VM is critical to ensuring uninterrupted operations. Possible denial-of-service (DoS) vectors or unexpected behavior could lead to operational disruptions, affecting data request execution and overall system stability.

Mitigation

Target each of the discussed wasm instructions and apply more granular gas cost measurements

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-wasm-vm/pull/59>

gjaldon

This [fix](#) limits the memory for both the host and the VM to 10MB (160 pages).

Issue H-5: Malicious WASM program can cause denial of service attack against SEDA validators through unbounded stdout/stderr

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/206>

Found by

tallo

Summary

The absence of size limits on WASM stdout/stderr outputs will cause a denial of service vulnerability for SEDA validators as attackers can exhaust node memory by creating data requests with WASM modules that generate excessive terminal output.

Root Cause

In `runtime.rs` there is no size limit enforcement for stdout/stderr buffers, unlike the explicit limit that exists for VM execution results:

```
// Add size check for execution result
if execution_result.len() > MAX_VM_RESULT_SIZE_BYTES {
    stderr.push(format!(
        "Result size ({} bytes) exceeds maximum allowed size ({} bytes)",
        execution_result.len(),
        MAX_VM_RESULT_SIZE_BYTES
    ));
    return Err(VmResultStatus::ResultSizeExceeded);
}
```

Internal Pre-conditions

1. Attacker needs to submit a data request with a custom WASM program that writes excessive amounts of data to stdout/stderr

External Pre-conditions

None

Attack Path

1. Attacker creates a malicious WASM module that repeatedly writes large chunks of data to stdout and stderr, potentially using a loop to generate several gigabytes of output
2. Attacker submits this module as part of a legitimate data request to the SEDA network with sufficient gas limit
3. When validators process this request during the tally phase in endblock.go, the WASM module executes and generates excessive stdout/stderr output
4. These outputs are collected without size limits and passed through multiple layers of memory allocation and copying
5. The outputs are then included in blockchain events, and written to consuming substantial memory and processing resources
6. Validator nodes experience memory exhaustion, potentially causing crashes and significant performance degradation

Impact

To cause the most impact a malicious user can submit multiple data requests to the same malicious program, these can all be processed in the same batch. Each stdout/stderr will be stored in memory: <https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-chain/x/tally/keeper/endblock.go#L80>

```
tallyResults := make([]TallyResult, len(tallyList))
dataResults := make([]batchingtypes.DataResult, len(tallyList))

for i, req := range tallyList {
    // [...]
    _, tallyResults[i] = k.FilterAndTally(ctx, req, params, gasMeter)
    // [...]
}
```

And allows for a max of u32 (4gb) according to the wasmer fd_write syscall: https://github.com/wasmerio/wasmer/blob/475f335cb5a84ef6a8179699d322ca776c1bd26b/lib/wasix/src/syscalls/wasi/fd_write.rs#L11C1-L32C32

```
/// ### `fd_write()`
/// Write data to the file descriptor
/// Inputs:
/// - `Fd`
///     File descriptor (opened with writing) to write to
/// - `const __wasi_ciovec_t *iovs`
///     List of vectors to read data from
/// - `u32 iovs_len`
///     Length of data in `iovs`
```

```

/// Output:
/// - `u32 *nwritten`
///     Number of bytes written
/// Errors:
///
#[instrument(level = "trace", skip_all, fields(%fd, nwritten = field::Empty), ret)]
pub fn fd_write<M: MemorySize>(
    mut ctx: FunctionEnvMut<'_, WasiEnv>,
    fd: WasiFd,
    iovec: WasmPtr<__wasi_ciovec_t<M>, M>,
    iovec_len: M::Offset,
    nwritten: WasmPtr<M::Offset, M>,
) -> Result<Errno, WasiError> {

```

meaning a single data request with stderr/stdout of 8gb total will be copied across rust strings, converted to Vec, serialized to C-compatible FFI structures, copied to Go data structures via CGO, formatted and joined for event attributes all while being stored in tallyResults. Since this is all executed in Endblocker, and multiple malicious data requests can be submitted, all validators will face extremely degraded performance and OOM crashes

Mitigation

Implement size limits for stdout and stderr similar to those already in place for execution results

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-wasm-vm/pull/65>

gjaldon

This fix limits the stdout and stderr to a configurable maximum amount of bytes.

Issue H-6: Attacker can exploits batch sender role to block result Submissions via fee transfer reversion

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/208>

Found by

Schnilch, newspacexyz, rsam_eth, zxripter

Summary

The `postBatch` function can be called by any address if all parameters are valid—that is, if the batch includes valid signatures from validators present in `s.lastValidatorsRoot` and the batch itself is valid. The address that submits the batch becomes its sender and is entitled to receive the `batchFee` rewards from all results corresponding to that batch. However, if the batch sender is a contract that reverts when receiving native tokens, then the results for that batch cannot be posted.

Root Cause

The `postBatch` function allows any address to submit a batch as long as the following conditions are met:

1. Batch Height is Greater Than the Previous One

```
if (newBatch.batchHeight <= s.lastBatchHeight) {
    revert InvalidBatchHeight();
}
```

which is true for upcoming batch

2. Validators in the `validatorProofs` Are Part of the Latest `lastValidatorsRoot`

```
if (!_verifyValidatorProof(validatorProofs[i], s.lastValidatorsRoot)) {
    revert InvalidValidatorProof();
}
```

This validation confirms that each validator who signed the batch is included in the most recent validators root.

3- signatures for `batchId` are valid and belong to the validators:

```
if (!_verifySignature(batchId, signatures[i], validatorProofs[i].signer)) {
    revert InvalidSignature();
}
```

```
}
```

4- voting power percentage exceeds the CONSENSUS_PERCENTAGE:

```
if (votingPower < CONSENSUS_PERCENTAGE) {  
    revert ConsensusNotReached();  
}
```

This check ensures that the cumulative voting power of the validators who signed the batch exceeds the consensus threshold (typically 66.6%).

Once these validations pass, the state is updated to assign the batch sender to `msg.sender`: <https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-evm-contracts/contracts/provers/Secp256k1ProverV1.sol#L131>

```
s.batches[newBatch.batchHeight] = BatchData({resultsRoot: newBatch.resultsRoot,  
↪ sender: msg.sender});
```

When `postResult` is later called, after verifying that the result ID is included in the batch's `resultsRoot`, the `batchFee` is sent to the `batchSender`:

<https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-evm-contracts/contracts/core/SedaCoreV1.sol#L181-L190>

```
if (requestDetails.batchFee > 0) {  
    if (batchSender == address(0)) {  
        // If no batch sender, send all batch fee to requestor  
        refundAmount += requestDetails.batchFee;  
    } else {  
        // Send batch fee to batch sender  
        // @audit batchSender does not accept ether  
        _transferFee(batchSender, requestDetails.batchFee);  
        emit FeeDistributed(result.drId, batchSender, requestDetails.batchFee,  
↪ ISedaCore.FeeType.BATCH);  
    }  
}
```

If the `batchSender` is a contract that does not accept native tokens, the fee transfer (and `postResult`) will fail: <https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-evm-contracts/contracts/core/SedaCoreV1.sol#L356-L360>

```
function _transferFee(address recipient, uint256 amount) internal {  
    // Using low-level call instead of transfer()  
    (bool success, ) = payable(recipient).call{value: amount}("");  
    if (!success) revert FeeTransferFailed();  
}
```

Internal Pre-conditions

- The contract must not be in a paused state.
- `Batch.resultsRoot` must not be `bytes32(0)`, meaning it must contain valid results.

External Pre-conditions

- The `postBatch` transaction is processed through a public mempool.
- The sender of `postBatch` is a contract that does not accept ether in its receive function.

Attack Path

- A very popular DeFi platform relies on the SEDA chain to fetch critical, time-sensitive data and submits 10 requests to the `SedaCoreV1` contract.
- A solver submits a `postBatch` transaction containing a `resultsRoot` that covers the results for these 10 requests, along with results for other requests.
- An attacker deploys a contract that reverts when it receives native tokens.
- The attacker front-runs the solver's transaction by offering a much higher gas fee.
- The attacker's transaction is processed, causing the malicious contract to become the batch sender.
- When `postResult` is called, it reverts because the batch sender contract does not accept native ens

Impact

- Results for the batch cannot be posted.
- The attack is almost cost-free for the attacker (aside from a negligible gas fee on L2s). An attacker can repeatedly become the batch sender for all batches and block all `postResult` requests.

PoC

No response

Mitigation

Ensure that batch sender is a valid solver:

```
function postBatch(  
    SedaDataTypes.Batch calldata newBatch,  
    bytes[] calldata signatures,  
    SedaDataTypes.ValidatorProof[] calldata validatorProofs  
) external override(ProverBase) whenNotPaused {  
    if(!validSubmitter[msg.sender]) revert InvalidBatchSender();
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-evm-contracts/pull/96>

gjaldon

This [PR](#) implements a pull-based model for fees where recipients withdraw their fees from the [SedaFeeManager](#). Any invalid recipients can no longer block `SedaCore` operations.

Issue H-7: Use of Vulnerable IBC-Go v8.4.0 - Non-Deterministic JSON Unmarshalling Can Cause Chain Halt

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/222>

Found by

x0lohacllohell

Summary

The protocol is using IBC-Go v8.4.0, which contains a critical vulnerability ([ASA-2025-004](#)) in the deserialization of IBC acknowledgements. This flaw results in non-deterministic behavior, which can lead to a chain halt if an attacker opens an IBC channel and sends a specially crafted acknowledgement packet.

Since any user with permission to open an IBC channel can exploit this issue, the vulnerability has an almost certain likelihood of occurrence, making it a critical security risk.

Root Cause

Usage of [IBC-Go v8.4.0](#),

Impact

An attacker can halt the chain by introducing a malformed acknowledgement packet.

Mitigation

Upgrade to the latest patched version of IBC-Go: v8.6.1

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/523>

gjaldon

Issue is fixed by upgrading to [Cosmos-SDK v0.50.13](#).

Issue H-8: Attacker can front-run Withdraw and steal the withdrawal

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/231>

Found by

000000, 0xeix, Boy2000, DeLaSoul, cu5t0mPe0, dod4ufn, g, leopoldflint, zxriotor

Summary

The Withdraw message in the Seda Core contract sends the withdrawn tokens to the message sender. This enables anyone to front-run the withdrawal with the same message to steal the withdrawn amount.

Root Cause

- The withdrawn tokens are sent to the message sender.

```
let bank_msg = BankMsg::Send {  
  to_address: info.sender.to_string(),  
  amount:    coins(self.amount.u128(), token),  
};
```

- No checks are done on the message sender, so it can be anyone.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Attacker observes a Withdraw message with valid proof.
2. Attacker frontruns the Withdraw copying the same message to steal the withdrawn amount.

Impact

Permanent loss of funds for the Staker.

PoC

No response

Mitigation

Consider sending the withdrawn tokens to a pre-approved address instead of the `info.sender`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain-contracts/pull/272>

gjaldon

It is no longer possible to front-run and steal withdrawal because the `withdraw_address` is now included in the signature.

Issue H-9: A jailed validator with no registered key blocks proving scheme activation

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/239>

Found by

g, zxriptor

Summary

When a proving scheme is ready for activation, all validators without registered keys will be jailed. However, there is no check that a validator is currently jailed before jailing, which raises an error and causes the `EndBlock()` to return early without activating the proving scheme.

Root Cause

- In Pubkey module's `EndBlock()`, a validator without keys will get jailed without first checking if it is already jailed.
- `slashingKeeper.Jail()` eventually calls `jailValidator()`, which returns an error when a validator is already jailed.

```
if validator.Jailed {  
    return types.ErrValidatorJailed.Wrapf("cannot jail already jailed validator,  
↪ validator: %v", validator)  
}
```

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. A validator has no registered keys. It is optional to register keys while the scheme is not activated.
2. This validator with no keys gets permanently Jailed for double-signing.

3. The Pubkey module's `EndBlock()` always returns early, because `JailValidators()` always fails.

Impact

The Proving Scheme will not be activated at the configured activation height and will remain inactive while the validator is jailed and has no registered key. A validator can be jailed permanently, leading to the Proving Scheme never getting activated.

A validator can prevent batches from ever getting produced because the `SEDAKeyIndexSecp256k1` proving scheme never gets activated.

PoC

None

Mitigation

Consider checking first if the Validator is Jailed before jailing it.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/522>

gjaldon

This issue has been fixed by no longer failing when validator is already jailed.

Issue H-10: Malicious proposer can submit a request with large invalid transactions because of no mempool to bloat the block store

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/241>

Found by

Boy2000, bronze_pickaxe, g

Summary

Context:

- Raw transaction bytes of every transaction included in a proposed block will be stored by CometBFT in its Blockstore
- There is no BlockStore pruning done by SEDA Chain

The SEDA Chain's proposal handlers are configured with a no-op mempool. This default `ProcessProposal` handler is called after the vote extensions have been verified. Since the handler is configured with a no-op mempool, no additional processing is done and the transactions are accepted with verification.

Given the above, a malicious proposer can abuse the lack of transaction validation and fill every block they propose with invalid transactions up to the `MaxBlockSize` (a consensus parameter set in CometBFT).

Root Cause

In [app.go:1012](#), the default proposal handlers are configured to use the `NoOpMempool`. However, it is not advisable to use that in production because of the lack of transaction verification.

```
defaultProposalHandler := baseapp.NewDefaultProposalHandler(mempool.NoOpMempool{}),  
↪ bApp)
```

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. A proposer proposes a block with a valid first transaction and the rest of the block filled with invalid transactions up to the `MaxBlockSize`.
2. This block with mostly invalid transactions will be accepted by every validator even if all the other transactions fail. The raw transaction bytes of all the transactions will be recorded in the `BlockStore`.

Impact

Permanent storage of invalid transactions that would bloat the chain unnecessarily and consume more resources during:

- Block propagation (network bandwidth)
- Block processing (CPU/memory)
- Block storage (disk space)
- Future node synchronization

PoC

None

Mitigation

Consider replacing `NoOpMempool` with a valid mempool when configuring the proposal handlers.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/520>

gjaldon

The mempool is now the `SenderNonceMempool` instead of the `NoOpMempool`. This fixes the issue.

Issue H-11: Tallying a Data Request with a wildcard expression in its consensus filter will store non-deterministic data and cause a chain halt

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/245>

Found by

g

Summary

In Tally module's `EndBlock()`, all tallying Data Requests will be processed. Each Data Request can have a Consensus Filter, which will be applied to every reveal object in the Data Request.

When the filter type is FilterStdDev or FilterMode, the consensus filter will be applied. The filter is treated as a path expression for querying data from a JSON object. One of the supported path expressions is the wildcard expression, which gets all the elements in the JSON object, but the results have a non-deterministic order. Due to this non-deterministic order, validators will get different `dataList`, `freq`, and `maxFreq`. This leads to a state divergence that will cause consensus failures, and ultimately, a chain halt.

Root Cause

In `filters_util.go:36-51`, any path expression is accepted and it expects that the results will have a deterministic ordering. Only the 0th-index of `elems` is accessed.

```
obj, err := parser.Parse(revealBytes)
if err != nil {
    errors[i] = true
    continue
}
expr, err := jp.ParseString(dataPath)
if err != nil {
    errors[i] = true
    continue
}
// @audit the path expression is applied here to query elements from the reveal JSON
↪ object
elems := expr.GetNodes(obj)
if len(elems) < 1 {
    errors[i] = true
    continue
}
```

```

}
// @audit only the first element is returned as data
data := elems[0].String()

```

Below is an example of a wildcard expression used to query a JSON object.

The results of applying the filter will be in the form of outliers and consensus, which are a []bool and bool.

```

outliers, consensus := filter.ApplyFilter(reveals, res.Errors)

```

outliers and consensus can be different values for different validators. These values affect the output data results, which will be stored.

```

_, tallyResults[i] = k.FilterAndTally(ctx, req, params, gasMeter)
// @audit the Result, ExitCode, and Consensus can be different across
↪ validators because of the non-deterministic
// results of applying the filter
dataResults[i].Result = tallyResults[i].Result
dataResults[i].ExitCode = tallyResults[i].ExitCode
dataResults[i].Consensus = tallyResults[i].Consensus
// ... snip ...
}

processedReqs[req.ID] = k.DistributionsFromGasMeter(ctx, req.ID, req.Height,
↪ gasMeter, params.BurnRatio)

dataResults[i].GasUsed = gasMeter.TotalGasUsed()
dataResults[i].Id, err = dataResults[i].TryHash()
// ... snip ...

// Store the data results for batching.
for i := range dataResults {
    // @audit the data results are stored, but since the data will be different
    ↪ across validators, there will be
    // state divergence.
    err := k.batchingKeeper.SetDataResultForBatching(ctx, dataResults[i])
}

```

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

A malicious user can abuse this.

1. A malicious user can post multiple valid Data Requests with a wildcard expression \$.* as consensus filter.
2. Once the valid Data Requests are in "Tallying Status", the Tally module will process them and store their corresponding Data Results for batching. Every validator here will store different values for their data results.
3. There will be a Chain Halt because there will be no consensus on the state root across validators.

Impact

Chain halt due to state divergence across validators.

PoC

None

Mitigation

Consider always sorting the result of `expr.GetNodes(obj)` before getting the first element as the result.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/525>

gjaldon

The issue has been fixed as recommended by sorting the result of `expr.GetNodes(obj)`.

Issue H-12: Attackers can flood validators with Commit/Reveal execution messages to delay blocks or DOS the node

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/246>

Found by

g, tallo

Summary

Commit and Reveal execution messages sent to the SEDA Core Contract are not charged any gas fees. This provides a way for malicious actors to DOS nodes or at the least delay blocks.

Root Cause

Before a transaction is executed, the AnteHandler is run. It checks if all of the transactions' messages is eligible for free gas and sets gas price to 0 when they are.

A message is eligible for free gas when it is a CommitDataResult or a RevealDataResult, and the executor can commit or reveal.

```
switch contractMsg := contractMsg.(type) {
case CommitDataResult:
    result, err := d.queryContract(ctx, coreContract,
    ↪ CanExecutorCommitQuery{CanExecutorCommit: contractMsg})
    if err != nil {
        return false
    }

    return result
case RevealDataResult:
    result, err := d.queryContract(ctx, coreContract,
    ↪ CanExecutorRevealQuery{CanExecutorReveal: contractMsg})
    if err != nil {
        return false
    }

    return result
}
```

A malicious user can abuse this unmetered execution by filling a transaction with *the same* CommitDataResult or RevealDataResult message.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Attacker sends a Transaction filled with the same CommitDataResult message for a data request that is ready for commitment.
2. Since the attacker's transaction passes `checkFreeGas()` for all its messages, the transaction will be eligible for free gas.

Multiple attackers can repeat this attack to exploit unmetered execution and unnecessarily consume validator resources.

Impact

This can cause chain delays or chain halts.

PoC

None

Mitigation

Consider checking that the transaction does not contain duplicate messages before making it eligible for free gas. Another option is to charge gas up front and provide a refund mechanism instead.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/527>

gjaldon

This [PR](#) implements a refund mechanism and ensures that CommitReveal messages can not be mixed with other messages and that they are unique. Refunds are triggered by the Core Contract sending a [RefundTx](#) message.

Issue H-13: Anyone can crash validators with a Tally VM program that panics the `call_result_write` import

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/247>

Found by

g

Summary

The Tally VM comes with imports that serve as a bridge between the host and the VM. These imports are run in the host environment. When an import panics, it crashes the host, which causes the validator node to crash. The `call_result_write` import can be used to trigger an out-of-range index access and panic. Since the tally programs are executed from the Tally module's `Endblock()`, every validator will crash when the described program is executed, which leads to a chain halt.

Root Cause

In the imported `call_result_write` function, the user can pass an arbitrary `result_data_length`. As long as `result_data_length` is a value greater than the length of `ctx.call_result_value`, the host environment will panic due to out-of-range index access.

```
fn call_result_value(
    env: FunctionEnvMut<'_, VmContext>,
    result_data_ptr: WasmPtr<u8>,
    result_data_length: u32,
) -> Result<()> {
    let ctx = env.data();
    let memory = ctx.memory_view(&env);

    let target = result_data_ptr.slice(&memory, result_data_length)?;
    let call_value = ctx.call_result_value.read();

    for index in 0..result_data_length {
        // @audit call_value[index] will panic when index is out of range. The user
        ↪ can easily trigger this.
        target.index(index as u64).write(call_value[index as usize])?;
    }

    Ok(())
}
```

The length of the default value of `ctx.call_result_value` is 0.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. The Attacker (anyone) create a WASM program that exploits the vulnerability in the `call_result_write` import.
2. The Attacker then compiles the WASM program and deploys the binary via `StoreOracleProgram()`.
3. The Attacker posts a Data Request that will execute their earlier deployed Tally Program.
4. Once the Data Request is for tallying, it will be processed in the Tally module's `EndBlock()`.
5. When executing the Attacker's Tally Program, it panics and crashes all the validators that run it. This causes a chain halt.

Impact

All validators that execute the Attacker's Tally program will crash and the SEDA Chain will halt.

PoC

The following WASM program will crash the host environment for the Tally VM.

```
#[link(wasm_import_module = "seda_v1")]
extern "C" {
    pub fn call_result_write(result: *const u8, result_length: u32);
}

fn main() {
    unsafe {
        call_result_write(result.as_ptr(), 1 as u32);
    }
}
```

To compile and run the above program, do the following:

1. Create the following Cargo.toml.

```
[package]
name = "attack"
version = "0.1.0"
edition = "2021"

[[bin]]
name = "attack"
path = "src/main.rs"
```

2. Run the following CLI commands.
3. Copy the output WASM file into the seda-wasm-vm directory. For example:
4. Add the below test to libtallyvm/src/lib.rs:

```
#[test]
fn execute_attack() {
    let wasm_bytes = include_bytes!("../../*.wasm");
    let mut envs: BTreeMap<String, String> = BTreeMap::new();

    envs.insert("VM_MODE".to_string(), "dr".to_string());
    envs.insert(DEFAULT_GAS_LIMIT_ENV_VAR.to_string(),
        ↪ "3000000000000000".to_string());

    let tempdir = std::env::temp_dir();
    let result = _execute_tally_vm(
        &tempdir,
        wasm_bytes.to_vec(),
        vec![],
        envs,
    )
    .unwrap();

    println!("Result: {:?}", result);
}
```

5. Run the test with `cargo test execute_attack`.

The test will crash with the following logs:

Mitigation

Consider changing the method of writing to `call_result_value` to something like:

```
let mut call_value = ctx.call_result_value.write();  
call_value.extend_from_slice(&target);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:

<https://github.com/sedaprotocol/seda-wasm-vm/pull/66>

gjaldon

The panics caused by out-of-bounds index access in the `call_result_write` import are no longer possible because of the fix introduced [here](#) that ensures that the `target` and `call_value` have the same length as `result_data_length`.

Issue H-14: `call_result_write` import can be exploited for unmetered execution and memory growth

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/248>

Found by

g, tallo

Summary

Unlike the other core Tally imports, `call_result_write` does not call `apply_gas_cost` to charge the caller gas. Any attacker can exploit this issue to do unmetered execution or memory growth because there is no cost to the attacker. This impacts validators by draining their resources, whether CPU or memory, and can lead to chain delays or Chain Halts, in the worst case.

Root Cause

In the `call_result_write` import, there is no call to `apply_gas_cost` unlike in the other imports.

```
fn call_result_value(
    env: FunctionEnvMut<'_, VmContext>,
    result_data_ptr: WasmPtr<u8>,
    result_data_length: u32,
) -> Result<()> {
    // @audit apply_gas_cost() must be called at the start to apply metering
    let ctx = env.data();
    let memory = ctx.memory_view(&env);

    let target = result_data_ptr.slice(&memory, result_data_length)?;
    let call_value = ctx.call_result_value.read();

    for index in 0..result_data_length {
        target.index(index as u64).write(call_value[index as usize])?;
    }

    Ok(())
}
```

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. The Attacker creates a WASM program that exploits the unmetered execution of `call_result_write`. The program can either loop for as long as there is gas available (they only need to pay for the gas for the loop) or set `result_data_length` to 100GB in length.
2. The Attacker then compiles the WASM program and deploys the binary via `StoreOracleProgram()`.
3. The Attacker posts a Data Request that will execute their earlier deployed Tally Program.
4. Once the Data Request is for tallying, it will be processed in the Tally module's `EndBlock()`.
5. When executing the Attacker's Tally Program, it either crashes all the validators that run it due to Out-of-Memory and cause a chain halt, or the unmetered execution delays block building significantly.

Impact

Unmetered execution or memory growth will drain the resources of Validators, leading to chain delays or chain halts.

PoC

None

Mitigation

Consider calling `apply_gas_cost` in the `call_result_write` import to apply metering.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-wasm-vm/pull/67>

gjaldon

Reading the `call_result_value` in the `call_result_write` import will now consume it. That means that calling the import will only work when `call_result_value` has data stored in it.

Issue H-15: WASI imports can be exploited for un-metered execution or unbounded memory growth

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/249>

Found by

g

Summary

All the WASI imports do not call `apply_gas_cost()`, so they do not do any metering. For example, `fd_write` does not do any metering. Any attacker can exploit this unmetered execution because there is no cost to the attacker. This impacts validators by draining their resources, whether CPU or memory, and can lead to chain delays or Chain Halts, in the worst case.

Root Cause

WASI objects are imported as-is direct from the WASI environment.

```
pub fn create_wasm_imports(
    store: &mut Store,
    vm_context: &FunctionEnv<VmContext>,
    wasi_env: &WasiFunctionEnv,
    wasm_module: &Module,
    call_data: &VmCallData,
) -> Result<Imports> {
    // @audit the WASI environment's import objects
    let wasi_import_obj = wasi_env.import_object(store, wasm_module)?;
    // ... snip ...

    for allowed_import in allowed_imports.iter() {
        // "env" is all our custom host imports
        if let Some(found_export) = custom_imports.get_export("seda_v1",
↪ allowed_import) {
            allowed_host_exports.insert(allowed_import.to_string(), found_export);
        } else if let Some(wasi_version) = wasi_version {
            // When we couldn't find a match in our custom import we try WASI
↪ imports
            // WASI has different versions of compatibility so it depends how the
↪ WASM was
            // build, that's why we use wasi_verison to determine the correct export
            // @audit the WASI import objects are imported as-is
            if let Some(found_export) =
↪ wasi_import_obj.get_export(wasi_version.get_namespace_str(), allowed_import) {
```

```

        allowed_wasi_exports.insert(allowed_import.to_string(),
↪   found_export);
    }
}
}

```

Since the WASI import objects are imported as-is, they do not do any metering. These imported functions must be wrapped in a new that applies gas metering.

Below is a list of all the WASI imports.

```

"args_get",
"args_sizes_get",
"proc_exit",
"fd_write",
"environ_get",
"environ_sizes_get",

```

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. The Attacker creates a WASM program that exploits the unmetered execution of any WASI imports. The program will call a WASI import in a loop for as long as gas is available (they only need to pay for the gas for the loop).
2. The Attacker then compiles the WASM program and deploys the binary via StoreOracleProgram().
3. The Attacker posts a Data Request that will execute their earlier deployed Tally Program.
4. Once the Data Request is for tallying, it will be processed in the Tally module's EndBlock().
5. When executing the Attacker's Tally Program, the unmetered execution can delay block building significantly.

This unmetered execution can be compounded by performing the same attack with many data requests, which can cause greater delays in block building.

Impact

Delay block building, possibly to the point of chain halt.

PoC

None

Mitigation

Consider wrapping WASI imports in import objects that call `apply_gas_cost()`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:

<https://github.com/sedaprotocol/seda-wasm-vm/pull/58>

gjaldon

Every WASI import except `proc_exit` is now wrapped in a custom import object that calls `apply_gas_cost`.

Issue H-16: Anyone can pass any length to some Tally imports to inflate memory, induce OOM, and crash validators

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/250>

Found by

g

Summary

A Tally program can use Tally imports like `secp256k1_verify` and set its `message_length`, `signature_length`, and `public_key_length` to `max::u32` and bloat memory usage by 12GB.

Root Cause

There are 2 root causes for this issue:

1. There are no limits for `message_length`, `signature_length`, and `public_key_length`.

```
fn secp256k1_verify(
    mut env: FunctionEnvMut<'_, VmContext>,
    message: WasmPtr<u8>,
    message_length: i64,
    signature: WasmPtr<u8>,
    signature_length: i32,
    public_key: WasmPtr<u8>,
    public_key_length: i32,
) -> Result<u8> {
    apply_gas_cost(
        crate::metering::ExternalCallType::Secp256k1Verify(message_length as u64),
        &mut env,
    )?;

    let ctx = env.data();
    let memory = ctx.memory_view(&env);

    // Fetch function arguments as Vec<u8>
    // @audit using max::u32 for each of lengths will allocate a total memory of
    ↪ 12GB
    let message = message.slice(&memory, message_length as u32)?.read_to_vec()?;
    let signature = signature.slice(&memory, signature_length as
    ↪ u32)?.read_to_vec()?;
```

```
let public_key = public_key.slice(&memory, public_key_length as  
↪ u32)?.read_to_vec()?;
```

Below is the calculation for the memory use if maxU32 is used as length for all.

2. The gas cost for calling `secp256k1_verify` with the max lengths is only $\sim 4.3e13$ SEDA tokens. One whole SEDA token is $1e18$ and the current price of 1 whole SEDA token $\sim \$0.03019$. The cost of bloating all SEDA Chain validators' memory by 12GB is negligible.

Calculation for the `secp256k1_verify` gas cost:

- Note that the issue also exists in `keccak256` and `execution_result` but in lesser degrees.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. The Attacker creates a WASM program that calls the `secp256k1_verify` import with `max::u32` values for `message_length`, `signature_length`, and `public_key_length`.
2. The Attacker then compiles the WASM program and deploys the binary via `StoreOracleProgram()`.
3. The Attacker posts a Data Request that will execute their earlier deployed Tally Program.
4. Once the Data Request is for tallying, it will be processed in the Tally module's `EndBlock()`.
5. When executing the Attacker's Tally Program, the validator node's memory usage will inflate by ~ 12 GB. This can crash multiple validators due to Out-of-Memory.

Impact

Crashing multiple validators due to Out-of-Memory can cause chain halts.

PoC

The example WASM program that exploits the issue in `secp256k1_verify`.

```

#[link(wasm_import_module = "seda_v1")]
extern "C" {
    pub fn secp256k1_verify(
        message: *const u8,
        message_length: i64,
        signature: *const u8,
        signature_length: i32,
        public_key: *const u8,
        public_key_length: i32,
    ) -> u8;
}

fn main() {
    let result = vec![1, 2, 3];
    unsafe {
        secp256k1_verify(
            result.as_ptr(),
            u32::MAX as i64,
            result.as_ptr(),
            -1i32,
            result.as_ptr(),
            -1i32,
        );
    }
}

```

Mitigation

Consider limiting all the length parameters in all the Tally imports and/or increase the gas costs.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-wasm-vm/pull/59>

gjaldon

The VM instance is now limited to 10MB of memory (160 pages). This limit applies to both the host and the VM.

Issue H-17: A request poster can set gas_price to 1 and pay minimal fees for a lot of gas and drain validators' resources

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/256>

Found by

g

Summary

Due to a lack of validation on gas_price, anyone can post a request to the SEDA Core contract that will consume a lot of resources of validators.

Root Cause

In `post_request():9-102`, no validations are done on the gas_price and the amount of funds that are required from the poster is just the total gas limit * gas price.

```
let required = (Uint128::from(self.posted_dr.exec_gas_limit) +  
↳ Uint128::from(self.posted_dr.tally_gas_limit))  
  .checked_mul(self.posted_dr.gas_price)?;
```

Since there is no minimum gas_price, the request poster can set it to 1.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. A request poster will submit a valid data request with gas_price set to 1, 1 replication factor, tally gas limit set to the max.
2. A malicious validator commit-reveals a result for this data request and its status changes to "tallying".

3. In the Tally module's `EndBlock()`, all "tallying" data requests will be processed. When the malicious data request is processed, it executes the Tally Program with the tally gas limit.
4. The Tally Program executed can allocate the maximum memory by using imports and loop until the tally gas limit is reached to inflate memory usage and block execution as long as possible. The inflated memory use lasts as long as the tally program has not exited.

Impact

Inflated resource use on validators running the Tally `EndBlock()` will cause chain delays or chain halts in the worst case.

PoC

None

Mitigation

Consider requiring a minimum `gas_price` per data request.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:

<https://github.com/sedaprotocol/seda-chain-contracts/pull/279>

gjaldon

There is now a minimum gas price when posting a data request that prevents this issue from happening.

Issue H-18: ASA-2025-003: Groups module can halt chain when handling a malicious proposal

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/271>

Found by

4n0nx, Boy2000, HarryBarz, x0lohacllohell

Summary

cosmos sdk recently publish a bug (<https://github.com/cosmos/cosmos-sdk/security/advisories/GHSA-x5vx-95h7-rv4p>) and its mandatory for protocols to update

As its already has been published by the cosmos group but the protocol didnt update I consider it as a medium not high

Name: ASA-2025-003: Groups module can halt chain when handling a malicious proposal
Component: CosmosSDK
Criticality: High (Considerable Impact; Likely Likelihood per [ACMv1.2](#))
Affected versions: <= v0.47.15, <= 0.50.11
Affected users: Validators, Full nodes, Users on chains that utilize the groups module
Description

An issue was discovered in the groups module where a malicious proposal would result in a division by zero, and subsequently halt a chain due to the resulting error. Any user that can interact with the groups module can introduce this state. Patches

The new Cosmos SDK release [v0.50.12](#) and [v0.47.16](#) fix this issue. Workarounds

There are no known workarounds for this issue. It is advised that chains apply the update.
Timeline

```
February 9, 2025, 5:18pm PST: Issue reported to the Cosmos Bug Bounty program
February 9, 2025, 8:12am PST: Issue triaged by Amulet on-call, and distributed to
    ↳ Core team
February 9, 2025, 12:25pm PST: Core team completes validation of issue
February 18, 2025, 8:00am PST / 17:00 CET: Pre-notification delivered
February 20, 2025, 8:00am PST / 17:00 CET: Patch made available
```

This issue was reported to the Cosmos Bug Bounty Program by [dongsam](#) on HackerOne on February 9, 2025. If you believe you have found a bug in the Interchain Stack or would like to contribute to the program by reporting a bug, please see <https://hackerone.com/cosmos>.

If you have questions about Interchain security efforts, please reach out to our official communication channel at security@interchain.io. For more information about the

Interchain Foundation's engagement with Amulet, and to sign up for security notification emails, please see <https://github.com/interchainio/security>.

A Github Security Advisory for this issue is available in the Cosmos SDK [repository](#).

Root Cause

.

Internal Pre-conditions

.

External Pre-conditions

.

Attack Path

.

Impact

.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/517>

gjaldon

Upgrading to Cosmos-SDK [v0.50.12](#) fixes this issue.

Issue H-19: Missing Minimum Length Check in verifyBatchSignatures Allows Chain Halting Attack

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/273>

Found by

0xHammad, 0xNirix, dod4ufn, g

Summary

Missing validation of minimum vote extension length will cause a chain-halting panic for all network participants as a malicious validator will submit a vote extension smaller than 65 bytes, triggering a slice out-of-bounds panic during consensus.

Root Cause

In `verifyBatchSignatures` method at <https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-chain/app/abci/handlers.go#L383>, the function checks for maximum length of vote extensions but fails to verify the minimum required length of 65 bytes before attempting to slice the first 65 bytes from the extension:

```
// Only checks maximum length
if len(voteExtension) > MaxVoteExtensionLength {
    h.logger.Error("invalid vote extension length", "len", len(voteExtension))
    return ErrInvalidVoteExtensionLength
}

// This line will panic if voteExtension has fewer than 65 bytes
sigPubKey, err := crypto.Ecrecover(batchID, voteExtension[:65])
```

Internal Pre-conditions

NA

External Pre-conditions

NA

Attack Path

1. A malicious validator creates an invalid vote extension with fewer than 65 bytes

2. The validator submits this malformed vote extension during the consensus process
3. When `verifyBatchSignatures` is called during either `VerifyVoteExtensionHandler` or `ProcessProposalHandler`, it attempts to access `voteExtension[:65]`
4. This causes a slice out-of-bounds panic because the slice doesn't have 65 elements
5. The panic can halt the entire chain

Impact

The entire blockchain network suffers a complete service interruption. This is a critical denial-of-service vulnerability that a malicious validator can use to bring down all other validators.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/516>

gjaldon

This issue has been addressed by adding a minimum length check for the vote extension.

Issue M-1: Wrong amount of gas will be used in a certain case

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/20>

Found by

000000

Summary

Wrong amount of gas will be used in a certain case

Root Cause

When executing the tally program during the end blocker, we call `gas_meter::MeterExecutorGasDivergent()` when the gas reports are not uniform. There, we have the following code:

```
for i, gasReport := range gasReports {
    executorGasReport := gasMeter.CorrectExecGasReportWithProxyGas(gasReport)
    adjGasReports[i] = min(executorGasReport,
    ↪ gasMeter.RemainingExecGas()/uint64(replicationFactor))
    if i == 0 || adjGasReports[i] < lowestReport {
        lowestReporterIndex = i
        lowestReport = adjGasReports[i]
    }
}
```

We get the lowest report index and its respective gas report. Then, we have a special case upon going over the executors:

```
if i == lowestReporterIndex {
    gasUsed = lowestGasUsed
}
gasMeter.ConsumeExecGasForExecutor(executor, gasUsed)
```

We compute a special `lowestGasUsed` and use it for the gas consumption. However, this fails to account for the case where 2 of the gas reports are the same, in that case we will only use the lowest gas for the first one in the array, while the second (and every next one if any) will use the normal gas.

Internal Pre-conditions

1. The gas reports are not all the same but there are 2 or more which are the same

External Pre-conditions

No external pre-conditions

Attack Path

1. There are 3 gas reports, [10, 10, 20]
2. The lowest report will be 10 and the index will be 0
3. The computed `lowestGasUsed` will only be applied for the first executor, even though the 1st index is also the lowest report, these are 2 such reports
4. Incorrect gas will be consumed

Impact

Incorrect gas consumption.

PoC

No response

Mitigation

Have an array of indices with the lowest gas report value and use the lowest gas value for each of them.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/545>

gjaldon

This issue is fixed by using an entropy value for sorting reveals so that the `lowestReporterIndex` will not always be the one with the lost Proxy pubkey.

Issue M-2: Data requests queue can be DoS'ed

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/53>

Found by

zxripter

Summary

Posting data requests on the SEDA chain is permissionless and does not incur fees. This can be exploited by creating thousands of dummy and cheap data requests that will never be executed by executors but, once expire, they will bloat the processing queue.

Root Cause

To post a data request, a caller only needs to pay enough tokens to cover gas for the request processing:

```
let required = (Uint128::from(self.posted_dr.exec_gas_limit) +
↳ Uint128::from(self.posted_dr.tally_gas_limit))
  .checked_mul(self.posted_dr.gas_price)?;
if funds < required {
  return Err(ContractError::InsufficientFunds(
    required,
    get_attached_funds(&info.funds, &token)?,
  ));
};
```

However, `exec_gas_limit`, `tally_gas_limit`, and `gas_price` are input parameters specified by the caller hence they can be set low to result in charges of 1 or a few units of aseda tokens.

Such results will likely be ignored by executors as the rewards are negligent, however, they will be included in the processing (tallying) after the timeout period passed:

```
pub fn expire_data_requests(&self, store: &mut dyn Storage, current_height:
↳ u64) -> StdResult<Vec<String>> {
  // remove them from the timeouts and return the hashes
  let drs_to_update_to_tally = self.timeouts.remove_by_timeout_height(store,
↳ current_height)?;

  drs_to_update_to_tally
    .into_iter()
    .map(|hash| {
      // get the dr itself
```

```

        let dr = self.get(store, &hash)?;
        // update it to tallying
@>        self.update(store, hash, dr, Some(DataRequestStatus::Tallying),
↪        current_height, true)?;
        Ok(hash.to_hex())
    })
    .collect::<StdResult<Vec<_>>>>()
}

```

By creating thousands of such data requests an attacker can bloat a processing queue and prevent legitimate requests from processing in time. As can be seen from the following code snippet, only a limited set of data requests can be handled in a block:

<https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-chain/x/tally/keeper/endblock.go#L53-L57>

It is worth noting that even a negligible amount paid by the attacker for data request processing will be refunded to them after the processing pipeline is completed.

Attack Path

1. Attacker creates 1000s of data requests
2. All these requests time out
3. x/tally module processes only a single page limited to `params.MaxTalliesPerBlock` potentially consisting of the attacker's fake requests only.

Impact

1. Chain slowdown
2. Legitimate requests time out

PoC

No response

Mitigation

Introduce a data request fee or make a data request call permissioned to the whitelisted solvers

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain-contracts/pull/277>

gjaldon

Issue is fixed. A minimum gas price and gas limit are enforced to ensure every posted data request pays a minimum fee.

Issue M-3: Executors will get underpaid while excessive gas will be refunded to the requestor

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/80>

The protocol has acknowledged this issue.

Found by

zxriotor

Summary

Not accounting for outliers in execution gas usage may result in honest executors being underpaid, even if the gas limit is not exhausted, with the remaining amount refunded to the data requestor.

Root Cause

In `gas_meter.go:137` a maximum gas that an executor can receive (`gasUsed`) is capped by the `remainingExecGas` divided by the `replicationFactor`:

```
func MeterExecutorGasUniform(executors []string, gasReport uint64, outliers []bool,
↪ replicationFactor uint16, gasMeter *types.GasMeter) {
    executorGasReport := gasMeter.CorrectExecGasReportWithProxyGas(gasReport)
@> gasUsed := min(executorGasReport,
↪ gasMeter.RemainingExecGas()/uint64(replicationFactor))
    for i, executor := range executors {
        if outliers != nil && outliers[i] {
            continue
        }
        gasMeter.ConsumeExecGasForExecutor(executor, gasUsed)
    }
}
```

At the same time, gas consumption is not recorded for outliers. If the reported gas (`executorGasReport`) exceeds `gasMeter.RemainingExecGas() / uint64(replicationFactor)`, honest executors will be underpaid, even though `gasMeter.RemainingExecGas()` will not be zero after consumption is recorded.

Eventually, this unused residue will be refunded to the data requester instead of being paid to the executors.

Internal Pre-conditions

1. Remaining exec gas is not enough to cover the expenses of ALL executors.
2. There are outliers among executors.

External Pre-conditions

None

Attack Path

Consider the following scenario:

Remaining exec gas: 90,000 Replication factor: 10 Gas reported: 10,000 (uniform)
Outliers: 2

The `gasUsed` will be calculated as 90,000 (remaining gas) / 10 (replication factor) = 9,000

Therefore, 8 honest executors will receive 9,000 gas each totaling to 72,000. Residue exec gas: 18,000 (will be refunded to data requester).

Each validator is underpaid by 1,000 gas, even though there is enough remaining gas to fully compensate them and even refund 8,000 gas to the data requester.

Impact

Executors are underpaid.

Please note that a similar conceptual mistake exists in the [MeterExecutorGasDivergent](#) function.

PoC

No response

Mitigation

Reduce the replication factor by the number of outliers to achieve a fairer distribution:

```
func MeterExecutorGasUniform(executors []string, gasReport uint64, outliers []bool,
    ↪ replicationFactor uint16, gasMeter *types.GasMeter) {
    executorGasReport := gasMeter.CorrectExecGasReportWithProxyGas(gasReport)
    --- gasUsed := min(executorGasReport,
    ↪ gasMeter.RemainingExecGas()/uint64(replicationFactor))
    +++ gasUsed := min(executorGasReport,
    ↪ gasMeter.RemainingExecGas()/uint64(replicationFactor - len(outliers)))
```


Issue M-4: Tally VM startup costs not charged early enough

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/157>

Found by

zxriotor

Summary

The gas for activating the WASM VM is not charged early enough to prevent DoS attacks.

Root Cause

In [seda-wasm-vm/runtime/core/src/runtime.rs#L85-L93](#), the gas is charged for starting up VM based on the formula $(\text{GAS_PER_BYTE} * \text{args_bytes_total as u64}) + \text{GAS_STARTUP}$ where $\text{GAS_PER_BYTE} = 10_000$ and $\text{GAS_STARTUP} = 5_000_000_000_000$.

```
let args_bytes_total = call_data.args.iter().fold(0, |acc, v| acc + v.len());
// Gas startup costs (for spinning up the VM)
let gas_cost = (GAS_PER_BYTE * args_bytes_total as u64) + GAS_STARTUP;

if gas_cost < gas_limit {
    set_remaining_points(&mut context.wasm_store, &wasmer_instance, gas_limit -
↳ gas_cost);
} else {
    set_remaining_points(&mut context.wasm_store, &wasmer_instance, 0);
}
```

However, this occurs quite late in the process, right before the tally function call, after several computationally expensive operations - such as creating the runtime, instantiating the environment and module, and allocating memory - have already been executed.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

An attacker can post data requests with a gas limit sufficient for the data request and filtering phases but leaving zero or nearly zero gas for the tallying phase. This triggers the `internal_run_vm()` function to spin up the VM completely, only to stop execution immediately when the first metered operation is executed due to gas exhaustion. This makes such attacks cheaper than expected.

Impact

DoS attack, chain slowdown.

PoC

No response

Mitigation

Do not start tally phase execution if the remaining gas is not enough for the VM spin-up.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-wasm-vm/pull/68>

gjaldon

The fix does not start the VM instance if there is not enough gas provided for the VM startup cost.

Issue M-5: Signatures for the first batch will be rejected by VerifyVoteExtensionHandler

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/188>

Found by

zxripor

Summary

An incorrect check for the first batch number will result in all vote extensions with batch signatures being rejected.

Root Cause

In the `VerifyVoteExtensionHandler`, a signature is verified to ensure it was properly signed by an active validator:

```
err = h.verifyBatchSignatures(ctx, batch.BatchNumber, batch.BatchId,
    ↪ req.VoteExtension, req.ValidatorAddress)
if err != nil {
    h.logger.Error("failed to verify batch signature", "req", req, "err", err)
    return nil, err
}
```

To ensure the batch was signed with the correct key used by the validator at the time of signing, the public key record is retrieved from the validator tree entry of the previous batch:

```
valEntry, err := h.batchingKeeper.GetValidatorTreeEntry(ctx, batchNum-1, valOper)
if err != nil {
    if errors.Is(err, collections.ErrNotFound) {
        if len(voteExtension) == 0 {
            return nil
        }
        return ErrUnexpectedBatchSignature
    }
    return err
}
expectedAddr = valEntry.EthAddress
```

There is an inherent cold start issue: when the very first batch is signed, there is no previous record to verify the validator's key. In this case, the code defaults to using the current key as stored in the `x/pubkey` module:

```

if batchNum == collections.DefaultSequenceStart {
    pubKey, err := h.pubKeyKeeper.GetValidatorKeyAtIndex(ctx, valOper,
↳   utils.SEDAKeyIndexSecp256k1)
    if err != nil {
        return err
    }
    expectedAddr, err = utils.PubKeyToEthAddress(pubKey)
    if err != nil {
        return err
    }
} else {
    // ... skipped for brevity ...
}

```

However, the condition used to check whether it is the first batch is incorrect. The numbering starts from `collections.DefaultSequenceStart + 1` and only increments, meaning the condition `batchNum == collections.DefaultSequenceStart` will never be satisfied:

```

if !errors.Is(err, types.ErrBatchingHasNotStarted) {
    return types.Batch{}, types.DataResultTreeEntries{}, nil, err
}
newBatchNum = collections.DefaultSequenceStart + 1

```

This causes the code to follow the default branch, where the validator key is not found, leading to the extension vote being rejected.

Internal Pre-conditions

Happens for the first batch only.

External Pre-conditions

None

Attack Path

See the explanation in the Root Cause section

Impact

Inability to sign the first batch.

PoC

No response

Mitigation

The check for the first batch must be `if batchNum == collections.DefaultSequenceStart + 1`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/516>

gjaldon

This is fixed by setting the first batch number to `collections.DefaultSequenceStart` in batching's EndBlock which makes this case run for the first batch.

Issue M-6: Validators will not be able to sign first batch

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/192>

Found by

zxriotor

Summary

The check for a validator's presence in the validator Merkle tree of the *previous* batch will cause an error if the first batch is being processed.

Root Cause

During signing the batch, in `ExtendVoteHandler()` there is check for the validator being present in the validators tree of the previous (`batch.BatchNumber-1`) batch:

```
_, err = h.batchingKeeper.GetValidatorTreeEntry(ctx, batch.BatchNumber-1,
↪ h.signer.GetValAddress())
if err != nil {
    if errors.Is(err, collections.ErrNotFound) {
        h.logger.Info("validator was not in the previous validator tree - not
↪ signing the batch")
    } else {
        h.logger.Error("unexpected error while checking previous validator tree
↪ entry", "err", err)
    }
    return nil, err
}
```

However, when the first batch is being processed, there is no previous batch, so `GetValidatorTreeEntry()` will return an error, causing an early exit from `ExtendVoteHandler()` without signing the batch.

Internal Pre-conditions

Happens for the first batch only.

External Pre-conditions

None

Attack Path

See a Root Cause section.

Impact

Validators will skip signing the first data batch.

PoC

No response

Mitigation

The simplest way is to skip this check if the first batch is being processed.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/516>

gjaldon

This is fixed by setting the first batch number to `collections.DefaultSequenceStart` in batching's `EndBlock` and by skipping the check that the Validator signed the previous batch for the first batch.

Issue M-7: requestId has no unique parameters leading to different collisions

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/212>

Found by

Oxeix, Boy2000, PNS, blutorque, leopoldflint, rsam_eth, zxripor

Summary

The requestId parameter that's derived when posting a request is not unique and therefore can't be replicated again if needed or can be blocked by other users with front-running.

Root Cause

Currently, the requestId that's obtained by calling `deriveRequestId()` cannot be sent again with the same parameters if needed (for instance, when dealing with data price feeds) and can be blocked as well leading to undesired behavior for the entity that tries to post it.

Internal Pre-conditions

-

External Pre-conditions

An entity posts a data request on one of the supported chains.

Attack Path

- 1) There can be a situation where the same data request with exactly the same input parameters is needed to be sent (when dealing with price feeds for example) but it can't be done as the one of the parameters is needed to be adjusted each time
- 2) Users who are interested in blocking a data request can infinitely front-run the transactions and block the data request posting (this could lead to a situation where some users benefit from the stale prices used in some other protocol that can't fetch the data)

Impact

Data request collisions open up many attack surfaces including the situation where protocols can't fetch the data if an attacker decides to front-run them each time and blocking the requests (if there is a data request with the same id, the tx will revert)

PoC

Consider the current requestId derivation process:

<https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-evm-contracts/contracts/libraries/SedaDataTypes.sol#L122-139>

So the requestId depends on each of this parameters. And if the same one already exists, the new request with the same params will be blocked:

<https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-evm-contracts/contracts/core/abstract/RequestHandlerBase.sol#L42-44>

```
if (bytes(_requestHandlerStorage().requests[requestId].version).length != 0) {  
    revert RequestAlreadyExists(requestId);  
}
```

First of all, there is no any nonce parameter in the requestId meaning the entity that posts a request has to always change them to make a new hash which damages user experience. The most serious impact though is an ability of malicious users to front-run the requests and always create a request with the same values as there is no `msg.sender` involved here. This leads to a situation where blockchains or users can't even post a request and this can be essential to the protocol if it depends on the external oracles and the data can't be fetched in time leading to stale state.

Mitigation

Introduce some unique parameters like nonce and `msg.sender` address.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:

<https://github.com/sedaprotocol/seda-evm-contracts/pull/97/files>

gjaldon

The issue is addressed. Posting a data request with the same derived ID will no longer fail.

Issue M-8: Attackers can flood solvers with thousands of requests and prevent fee payouts

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/217>

Found by

000000, 0xMgwan, 0xlookman, Boy2000, Kodyvim, PASCAL, blutorque, g, gegul, leopoldflint, rsam_eth, zxripor

Summary

The current fee distribution model compensates solvers only after the result is posted on-chain—a fee system known as the push model. In this design, solver must call the `postResult` function to trigger fee payments. However, this mechanism can be exploited. An attacker can deploy tens of smart contracts that appear to submit attractive requests with generous fees, but are engineered to revert on native token transfers upon receiving the refund amounts.

```
// Example: Attacker switches the `revertOnReceiveEther` in their contract to true,
↳ to prevent a result from getting posted
if (refundAmount > 0) {
    _transferFee(requestDetails.requestor, refundAmount);
    emit FeeDistributed(result.drId, requestDetails.requestor, refundAmount,
↳ ISedaCore.FeeType.REFUND);
}
```

(these contracts may even include an on/off switch that bypasses solvers' checks for whether the requester can receive Ether). As a result, the network is forced to process these requests. Then, just before the results are posted, the attacker can disable Ether acceptance, thereby preventing fee transfers. Once the timeout period expires, the attacker can withdraw funds from all the requests using `withdrawTimedOutRequest`. In effect, the attacker incurs only minimal gas fees (especially on L2s) while overloading solvers and the Seda chain with tasks.

Root Cause

After a request is posted, its fees remain in the contract until results are posted using the `postResult` function. The fee distribution occurs only after solvers submit the results and the contract verifies that the `result ID` is included in the batch's `resultsRoot`.

For the malicious requester to trigger a revert during the native token transfer (thus blocking fee distribution), the `refundAmount` must be greater than zero. This can be achieved by setting the `request.gasLimit` slightly higher than the actual gas used, When

the fee is calculated: <https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-evm-contracts/contracts/core/SedaCoreV1.sol#L162>

```
// Split request fee proportionally based on gas used vs gas limit
//@audit if gasLimit > gasUsed, then submitterFee < requestDetails.requestFee
uint256 submitterFee = (result.gasUsed * requestDetails.requestFee) /
    ↪ requestDetails.gasLimit;
if (submitterFee > 0) {
    _transferFee(payableAddress, submitterFee);
    emit FeeDistributed(result.drId, payableAddress, submitterFee,
    ↪ ISedaCore.FeeType.REQUEST);
}
//@audit remaining amount to refund
refundAmount += requestDetails.requestFee - submitterFee;
```

Later, when transferring the refund to the requester:

```
// Aggregate refund to requestor containing:
// - unused request fees (when gas used < gas limit)
// - full request fee (when invalid payback address)
// - batch fee (when no batch sender)
if (refundAmount > 0) {
    _transferFee(requestDetails.requestor, refundAmount);
    emit FeeDistributed(result.drId, requestDetails.requestor, refundAmount,
    ↪ ISedaCore.FeeType.REFUND);
}
```

If the requester is a contract that reverts on receiving native tokens, the transfer will fail. For instance:

```
contract Malicious {
    //...rest of the code
    bool acceptEther;
    receive() external payable {
        if (!acceptEther) revert();
    }
    //...rest of the code
}
```

Finally, after the timeout period (`_storageV1().timeoutPeriod`), the attacker can withdraw the funds from all requests by calling `withdrawTimedOutRequest`.

Internal Pre-conditions

- Contract not be in paused state

External Pre-conditions

- Solvers are actively picking up requests with sufficiently attractive fees.
- Solvers are posting results back to SedaCoreV1 using the `postResult` function.
- The requester is a contract engineered to reject native token transfers (with an on/off switch for Ether reception), allowing later withdrawal of funds.

Attack Path

- The attacker deploys 100 different smart contracts with identical behavior (including an on/off feature for receiving Ether) on Optimism.
- Each malicious contract submits 100 requests with sufficiently high fees to attract solvers.
- Solvers pick up these requests one by one and relay them to the Seda chain.
- The requests are processed on the Seda network.
- The results are batched and posted on-chain.
- When a solver attempts to post the results to claim the fees, the attacker disables Ether reception in the malicious contracts.
- As a result, fee transfers to solvers revert.
- Solvers become congested with $100 \times 100 = 10,000$ processed requests that yield no fee payouts, overloading the system.

Impact

- **Uncompensated Solvers and SEDA Chain:**
The attack causes solvers to perform all necessary computations and batch processing without receiving any compensation.
- **Potential Network Instability:**
Over time, the inability to compensate solvers may lead to decline in the overall performance (since attacker can perform major request attacks each time from anonymous contracts) and trust in the blockchain ecosystem.

PoC

n/a

Mitigation

Instead of using a push model for fee distribution, a pull model should be adopted. In a pull model, all parties are credited with the appropriate amounts of native tokens within

the contract. They can then withdraw these funds at their discretion.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-evm-contracts/pull/96>

gjaldon

This PR implements a pull-based model for fees where recipients withdraw their fees from the SedaFeeManager. Any invalid recipients can no longer block SedaCore operations.

Issue M-9: Mean-Based Outlier Detection Vulnerability Allows Single Node to Sabotage Consensus

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/230>

Found by

0xNirix

Summary

Using the arithmetic mean instead of median in the standard deviation filter can cause a consensus failure for the entire protocol as a malicious nodes can manipulate the mean/standard deviation calculation by submitting appropriate values. And if that is not possible based on distribution of values, it can also strategically alter which values are considered part of the consensus set, thereby manipulating the final output value even if later median is used for the final calculation.

Root Cause

The choice to use arithmetic mean in the `FilterStdDev.ApplyFilter` at <https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-chain/x/tally/types/filters.go#L225> implementation is problematic as it makes the outlier detection mechanism vulnerable to extreme value manipulation. In `detectOutliersBigInt()`, the code calculates the mean of all values (`mean := sum.Div(sum, n)`) rather than the median, making the system susceptible to skewing by a single extreme value.

Notably, there is a **comment** in the code that suggests median should have been used:

```
// ApplyFilter applies the Standard Deviation Filter and returns an
// outlier list. A reveal is declared an outlier if it deviates from
// the median by more than the sample standard deviation multiplied
// by the given sigma multiplier value.
```

This comment explicitly states that values should be compared against the **median**, but the implementation actually compares them against the **mean**, creating potential for this vulnerability.

Internal Pre-conditions

NA

External Pre-conditions

NA

Attack Path

1. **Honest nodes submit values** within a reasonable range
2. **Malicious node submits an extreme value** close to the maximum allowed by the data type (e.g., close to `maxUint64`)
3. **Mean calculation is severely skewed** toward the extreme value
4. **Standard deviation calculation is affected**
5. **Some honest values are marked as outliers** since they deviate significantly from the skewed mean
6. **Consensus check fails** as less than 2/3 of nodes are considered non-outliers OR the new mean/median output is incorrect

Impact

The protocol suffers consensus failure, preventing it from finalizing the current round of data aggregation. The attacker gains the ability to block consensus without requiring control of 1/3 of the network, circumventing the protocol's fault tolerance guarantees. More importantly, even when consensus doesn't completely fail, the attacker can strategically manipulate which values are included in the final consensus set. This allows them to shift the final output value (regardless of whether mean or median is used for the final calculation) by selectively excluding honest node contributions that don't align with their desired outcome.

PoC

The vulnerability can be demonstrated using Chebyshev's inequality, which states: $P(|X - \mu| \geq k\sigma) \leq 1/k^2$ Where:

X is a random variable with mean μ and standard deviation σ k is a positive real number

According to Chebyshev's inequality, up to $1/k^2$ values can be outside k standard deviations from the mean. This means if sigma multiplier (k) is set to 1.5, up to 44% of values can be legitimately outside this range. A malicious node or group of malicious nodes can exploit this in two critical ways: a) They can strategically submit values that push up to 44% of values outside the acceptable range, causing consensus failure since consensus requires 2/3 (or approximately 67%) of nodes to agree. By pushing up to 44% of values outside the range, they ensure that less than the required threshold remains, effectively blocking consensus. b) Alternatively if above is not possible due to distribution, they can selectively target specific honest values to push out of the

consensus set while allowing others to remain. This strategic exclusion lets them manipulate the final output value, whether it's calculated as a mean or median, since they've already controlled which values are included in that calculation.

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/541>

gjaldon

The issue is fixed. The median is used instead of the mean when checking if a value is within max deviation.

Issue M-10: Anyone can post a request with gasPrice of 0 to cause SEDA chain to halt

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/235>

Found by

g

Summary

There are no validations preventing a data request's gasPrice to be zero. When the SEDA Chain tallies data requests, it uses gasPrice as a divisor in one of its gas calculations. A zero gasPrice leads to a division-by-zero that crashes all validator nodes.

Root Cause

- In SEDA Chain's Core contract, no validation is done on the value of the gasPrice when posting a request.
- In SEDA Chain's Tally module, it uses gasPrice as the divisor when metering the gas for proxies.

```
gasUsedPerExecInt := proxyConfig.Fee.Amount.Quo(gasMeter.GasPrice())
```

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. An attacker can post a valid request with gasPrice of 0, replicationFactor of 1, and no filters.
2. Once the request has been commit-revealed by a validator, whether maliciously or unwittingly, it will be processed by the SEDA Chain's Tally module Endblock().
3. Since no filtering is applied, there will be no filtering errors and it attempts to meter the gas for the proxies. The node will then panic and crash.

4. Every validator node will process this data request since it has a "Tallying" status and they will all crash until the issue is resolved.

Impact

This causes all validator nodes to crash indefinitely, leading to a Chain halt.

PoC

None

Mitigation

Consider validating that the `gasPrice` is non-zero before using it as a divisor or checking that `gasPrice` is a minimum non-zero value.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: <https://github.com/sedaprotocol/seda-chain/pull/530> <https://github.com/sedaprotocol/seda-chain-contracts/pull/277> <https://github.com/sedaprotocol/seda-evm-contracts/pull/98>

gjaldon

There is now a minimum gas price when posting a data request that prevents this issue from happening.

Issue M-11: Chain can deadlock due to no consensus because New Validators can not submit vote extensions

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/243>

The protocol has acknowledged this issue.

Found by

g

Summary

New validators will increase the total voting power, but their vote on vote extensions will not be counted for 1 block.

A validator's vote must be counted for the current block if they have an entry in a previous batch. When the following condition is true:

`PrepareProposal()` and `ProcessProposal()` will always fail when validating vote extensions and cause a chain halt.

Root Cause

In `abci/handlers.go#ExtendVoteHandler():101`, a new Validator will not be able to submit their vote extension because they do not yet have an entry in a previous batch.

```
// Check if the validator was in the previous validator tree.
// If not, it means the validator just joined the active set,
// so it should start signing from the next batch.
_, err = h.batchingKeeper.GetValidatorTreeEntry(ctx, batch.BatchNumber-1,
↳ h.signer.GetValAddress())
```

This check is also enforced in `VerifyVoteExtensionHandler()`, where a vote extension must be submitted by a validator with an entry in a previous batch.

Internal Pre-conditions

1. The total power of New Validators is greater than 1/3.

External Pre-conditions

None

Attack Path

1. All the vote extensions of New Validators are not submitted, which makes up for more than 1/3 of total voting power.
2. When the proposal is Prepared or Processed, `ValidateVoteExtensions()` will always fail because it requires consensus on vote extensions.

Impact

Chain halt because consensus will always fail on vote extensions no matter the proposer.

PoC

None

Mitigation

Consider modifying `ValidateVoteExtensions()` and filtering out the New Validators from being counted for TotalVotingPower.

Issue M-12: Anyone can front-run the creation of a vesting account to block it

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/251>

The protocol has acknowledged this issue.

Found by

000000, g

Summary

Creation of a vesting account will not work when the recipient of the vested funds already exists. Anyone can then block a new vesting account from being created by front-running it with a Bank::Msg.Send.

Root Cause

A vesting account can only be created if the recipient does not exist yet.

```
if acc := m.ak.GetAccount(ctx, to); acc != nil {  
    return nil, sdkerrors.ErrInvalidRequest.Wrapf("account %s already exists",  
↪ msg.ToAddress)  
}
```

This can be exploited by anyone that sends coins to the intended vesting recipient.

```
func (k BaseSendKeeper) SendCoins(ctx context.Context, fromAddr, toAddr  
↪ sdk.AccAddress, amt sdk.Coins) error {  
    // ... snip ...  
  
    // Create account if recipient does not exist.  
    //  
    // NOTE: This should ultimately be removed in favor a more flexible approach  
    // such as delegated fee messages.  
    accExists := k.ak.HasAccount(ctx, toAddr)  
    if !accExists {  
        defer telemetry.IncrCounter(1, "new", "account")  
        k.ak.SetAccount(ctx, k.ak.NewAccountWithAddress(ctx, toAddr))  
    }
```

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. A user calls `CreateVestingAccount()` with `pubkeyB` as the intended recipient.
2. The griever frontruns the `CreateVestingAccount()` transaction with Bank `Send()` transaction. This transaction creates the recipient account causing the `CreateVestingAccount()` to fail.

Impact

The intended recipient of the vesting funds is permanently blocked from becoming a Vesting Account. Once an account is created, it can not be deleted.

PoC

None

Mitigation

Consider updating an existing account to a Vesting Account.

Issue M-13: Executors/Proxies can game the rewards system by using a pubkey that will be sorted first

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/257>

Found by

g, zxriptor

Summary

The reveals and proxy pubkeys are sorted in ascending order every time data requests are filtered and tallied. The same ordering is used when metering gas for Proxies and Executors. Proxies and Executors that are sorted first will be prioritized for rewards distribution.

Root Cause

In `FilterAndTally():185-197`, the `reveals` are sorted in ascending order by their executor's key while the `ProxyPubkeys` of each reveal are also sorted in ascending order.

```
keys := make([]string, len(req.Reveals))
i := 0
for k := range req.Reveals {
    keys[i] = k
    i++
}
sort.Strings(keys)

reveals := make([]types.RevealBody, len(req.Reveals))
for i, k := range keys {
    reveals[i] = req.Reveals[k]
    sort.Strings(reveals[i].ProxyPubKeys)
}
```

When metering gas for proxies, the proxies are allocated gas in the same order until there is no remaining execution gas. Proxies with pubkeys that are ordered first will be prioritized.

The same behavior applies when metering gas for executors.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. A Proxy/Executor will generate and use public keys that will be ordered first when sorted.
2. When gas rewards are allocated and distributed, they will always be prioritized over others.

Impact

The rewards system can be gamed so that certain Executors/Proxies will always be prioritized for rewards even when all participating proxies/executors provide the same value.

PoC

None

Mitigation

Consider randomizing the order of the executors' and proxies' public keys when allocating their rewards.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/538>

gjaldon

Entropy is introduced when sorting the proxy pubkeys so that the order of the priority of rewards is random across different requests.

Issue M-14: The outlier gets the reduced payout when there is consensus on errors

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/284>

Found by

g

Summary

The payout for Executors is reduced when there is no consensus on the filter results. However, when 2/3 of the reveals are errors, it is not treated as consensus because of an error in the consensus check.

Root Cause

In filter.go:79, the consensus check for errors is incorrectly implemented. The errors must be greater than 2/3 of the reveals for a consensus with errors, which should not be the case.

```
// @audit the check must be `countErrors(res.errors)*3 >= len(reveals)*2`  
case countErrors(res.Errors)*3 > len(reveals)*2:
```

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. The Tally Module is processing a data request and executing its filter. It has 2/3 of the reveals as errors and has a `FilterMode` filter.
2. When the `FilterMode` filter is applied, it returns a slice of outliers with the errors set as outliers and false on consensus.
3. Since there is consensus on errors, this branch should be executed, and consensus should be set to true. Instead, the no-consensus branch runs.

4. When the metering for executors is processed, the reduced payout is applied because of the `ErrConsensusInError` error.

Impact

The outlier who reported no error will receive a reduced payout. The intended behavior is for the non-outliers who reported the errors to receive the full payout.

PoC

None

Mitigation

Consider changing the check to `case countErrors(res.Errors)*3 >= len(reveals)*2:`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-chain/pull/524>

gjaldon

The fix was applied as recommended.

Issue M-15: Malicious contracts can force excessive memory usage at minimal gas cost, threatening node stability and economic security

Source: <https://github.com/sherlock-audit/2024-12-seda-protocol-judging/issues/298>

Found by

stonejiajia

Summary

Incorrect implementation of WebAssembly memory growth gas metering will cause a resource exhaustion vulnerability for blockchain node operators as attackers will deploy contracts that allocate large amounts of memory while bypassing appropriate gas costs.

Root Cause

In `metering.rs` <https://github.com/sherlock-audit/2024-12-seda-protocol/blob/main/seda-wasm-vm/runtime/core/src/metering.rs#L56>

In the gas metering code for WebAssembly operations, specifically in the `get_wasm_operation_gas_cost` function:

```
Operator::MemoryGrow { mem, mem_byte: _ } => {  
    GAS_MEMORY_GROW_BASE + ((WASM_PAGE_SIZE as u64 * *mem as u64) * GAS_PER_BYTE)  
}
```

The implementation incorrectly uses the memory index (`mem`) instead of the actual number of pages being allocated. This is a fundamental misunderstanding of the WebAssembly specification, where:

1. The `mem` parameter refers to the memory index (which memory segment to operate on)
2. The actual number of pages to grow is a runtime value taken from the stack
3. In most WebAssembly modules, `mem` is 0 (since they only have one memory section)

This leads to almost all memory growth operations only being charged the base fee (`GAS_MEMORY_GROW_BASE`), regardless of how many pages are actually allocated.

Relevant specifications and references:

MDN documentation clearly states that the page count parameter for `memory.grow` is passed through the stack, as in `(memory.grow (i32.const 1))`.

<https://developer.mozilla.org/en-US/docs/WebAssembly/Reference/Memory/Grow>

The operational semantics of `memory.grow` in the WebAssembly Core Specification explicitly depends on the runtime stack top value (see Instructions section in Chapter 6).

<https://webassembly.github.io/spec/core/syntax/instructions.html>

The JavaScript API design of `WebAssembly.Memory.grow()` also verifies that the page count is a dynamic parameter https://developer.mozilla.org/en-US/docs/WebAssembly/Reference/JavaScript_interface/Memory/grow

Internal Pre-conditions

1. A WebAssembly module needs to include memory growth operations (`memory.grow`)
2. The blockchain system must be using this gas metering implementation
3. The contract must be accepted by the blockchain validators/nodes

External Pre-conditions

None identified.

Attack Path

1. Attacker creates a malicious WebAssembly contract that includes `memory.grow` operations requesting large amounts of memory (hundreds or thousands of pages)
2. Attacker deploys this contract to the blockchain
3. When the contract executes, it calls `memory.grow` with a large value
4. The VM charges only `GAS_MEMORY_GROW_BASE` gas (fixed cost) for this operation, regardless of the actual memory pages allocated
5. The contract can now use this large memory allocation for computation without having paid the appropriate gas costs
6. Attacker can repeat this process with multiple contracts, causing excessive memory consumption across blockchain nodes

Impact

The blockchain nodes suffer excessive resource consumption without corresponding gas payments. This could lead to:

1. Reduced node performance due to memory pressure
2. Potential denial of service for legitimate transactions
3. Breaking of the economic model intended to balance resource usage with gas costs
4. In extreme cases, nodes may crash due to out-of-memory conditions

The economic impact is significant as attackers can execute operations that should cost $\text{GAS_MEMORY_GROW_BASE} + (\text{WASM_PAGE_SIZE} * \text{actual_pages} * \text{GAS_PER_BYTE})$ but instead only pay $\text{GAS_MEMORY_GROW_BASE}$.

PoC

```
// Simple WebAssembly module (in text format) that exploits this vulnerability:
(module
  (memory 1)
  (func $exploit (result i32)
    ;; Try to grow memory by 1000 pages (65MB)
    i32.const 1000
    memory.grow
  )
  (export "exploit" (func $exploit))
)
```

This module would be charged only $\text{GAS_MEMORY_GROW_BASE}$ despite allocating 1000 pages (65MB) of memory.

Mitigation

The gas metering for memory growth operations should be implemented as a runtime check rather than static analysis. Two approaches are recommended:

1. Modify the metering middleware to inject runtime checks for memory growth:

```
// Pseudocode for runtime metering
if operation == memory.grow {
  let pages_to_grow = get_value_from_stack();
  let gas_cost = GAS_MEMORY_GROW_BASE + (WASM_PAGE_SIZE as u64 * pages_to_grow *
↪ GAS_PER_BYTE);
  charge_gas(gas_cost);
}
```

2. Implement a host function hook that intercepts memory.grow operations:

```
// Host function approach
fn intercept_memory_grow(&mut self, pages: u32) -> Result<i32, Error> {
  let gas_cost = GAS_MEMORY_GROW_BASE + (WASM_PAGE_SIZE as u64 * pages as u64 *
↪ GAS_PER_BYTE);
```

```
self.charge_gas(gas_cost)?;  
  
// Proceed with actual memory grow operation  
self.original_memory_grow(pages)  
}
```

The fix must be applied immediately as this vulnerability fundamentally breaks the economic security model of the blockchain's resource accounting.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/sedaprotocol/seda-wasm-vm/pull/59>

gjaldon

Limiting the memory allocated to the VM instance fixes this issue.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.