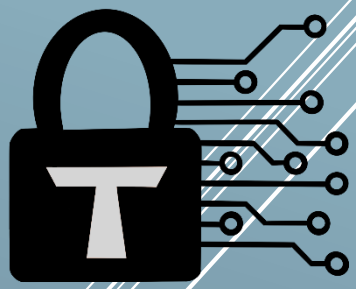
Trust Security

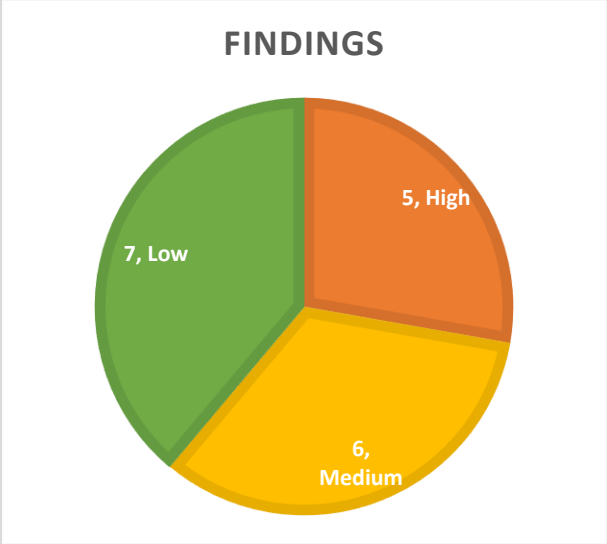


Blockchain Audit

Story Protocol – Blockchain Assets

13/12/2024

Executive summary

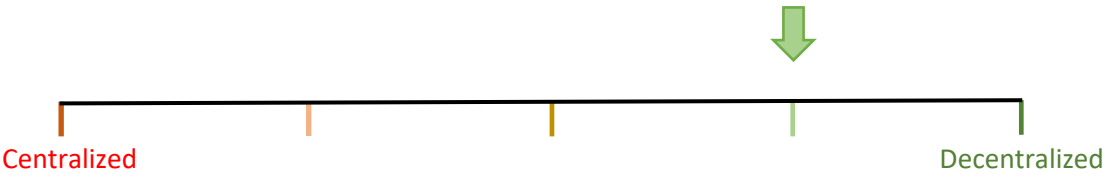


Category	L1 Blockchain
Audited file count	183
Lines of Code	19532
Auditor	christianvari Lambda
Time period	17/10/24 – 18/11/24

Findings

Severity	Total	Fixed	Acknowledged
High	5	4	1
Medium	6	5	1
Low	7	2	5

Centralization score



Signature

Digitally signed by Trust
Date: 2024.12.13 12:29:58 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	18.11.2024	Client report
0.2	11.12.2024	Removed guardian module
0.3	13.12.2024	Fix review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- story-geth/core/vm/contracts.go
- story-geth/core/vm/evm.go
- story-geth/crypto/secp256r1/publickey.go
- story-geth/crypto/secp256r1/verifier.go
- story-geth/params/protocol_params.go
- story/client/x/mint/types/params.go
- story/client/x/mint/types/keys.go
- story/client/x/mint/types/expected_keepers.go
- story/client/x/mint/types/events.go
- story/client/x/mint/types/query.proto
- story/client/x/mint/types/genesis.proto
- story/client/x/mint/types/genesis.go
- story/client/x/mint/types/codec.go
- story/client/x/mint/types/mint.proto
- story/client/x/mint/module/module.proto
- story/client/x/mint/module/depinject.go
- story/client/x/mint/module/module.go
- story/client/x/mint/keeper/params.go
- story/client/x/mint/keeper/keeper.go
- story/client/x/mint/keeper/grpc_query.go
- story/client/x/mint/keeper/abci.go
- story/client/x/mint/keeper/genesis.go
- story/client/x/evmstaking/types/withdraw.go
- story/client/x/evmstaking/types/params.go
- story/client/x/evmstaking/types/keys.go
- story/client/x/evmstaking/types/expected_keepers.go
- story/client/x/evmstaking/types/events.go
- story/client/x/evmstaking/types/staking_contract.go
- story/client/x/evmstaking/types/query.proto
- story/client/x/evmstaking/types/genesis.proto
- story/client/x/evmstaking/types/genesis.go
- story/client/x/evmstaking/types/codec.go
- story/client/x/evmstaking/types/evmstaking.proto
- story/client/x/evmstaking/types/params.proto

- story/client/x/evmstaking/module/module.proto
- story/client/x/evmstaking/module/depinject.go
- story/client/x/evmstaking/module/module.go
- story/client/x/evmstaking/keeper/deposit.go
- story/client/x/evmstaking/keeper/withdraw.go
- story/client/x/evmstaking/keeper/params.go
- story/client/x/evmstaking/keeper/singularity.go
- story/client/x/evmstaking/keeper/delegator_address.go
- story/client/x/evmstaking/keeper/validator.go
- story/client/x/evmstaking/keeper/keeper.go
- story/client/x/evmstaking/keeper/keys.go
- story/client/x/evmstaking/keeper/withdrawal_queue.go
- story/client/x/evmstaking/keeper/grpc_query.go
- story/client/x/evmstaking/keeper/update_commission.go
- story/client/x/evmstaking/keeper/redelegation.go
- story/client/x/evmstaking/keeper/reward_queue.go
- story/client/x/evmstaking/keeper/staking_queue.go
- story/client/x/evmstaking/keeper/utils.go
- story/client/x/evmstaking/keeper/ubi.go
- story/client/x/evmstaking/keeper/abci.go
- story/client/x/evmstaking/keeper/genesis.go
- story/client/x/evmstaking/keeper/unjail.go
- story/client/x/evmengine/types/tx.proto
- story/client/x/evmengine/types/upgrade_contract.go
- story/client/x/evmengine/types/params.go
- story/client/x/evmengine/types/keys.go
- story/client/x/evmengine/types/expected_keepers.go
- story/client/x/evmengine/types/events.go
- story/client/x/evmengine/types/ubi_contract.go
- story/client/x/evmengine/types/genesis.proto
- story/client/x/evmengine/types/genesis.go
- story/client/x/evmengine/types/cpayload.go
- story/client/x/evmengine/types/tx.go
- story/client/x/evmengine/types/codec.go
- story/client/x/evmengine/types/params.proto
- story/client/x/evmengine/module/module.proto
- story/client/x/evmengine/module/depinject.go
- story/client/x/evmengine/module/module.go
- story/client/x/evmengine/keeper/params.go
- story/client/x/evmengine/keeper/hooks.go
- story/client/x/evmengine/keeper/db.go
- story/client/x/evmengine/keeper/keeper.go
- story/client/x/evmengine/keeper/evmmsgs.go
- story/client/x/evmengine/keeper/proposal_server.go
- story/client/x/evmengine/keeper/ubi.go
- story/client/x/evmengine/keeper/msg_server.go

- story/client/x/evmengine/keeper/abci.go
- story/client/x/evmengine/keeper/genesis.go
- story/client/x/evmengine/keeper/evmengine.proto
- story/client/x/evmengine/keeper/evmengine.cosmos_orm.go
- story/client/x/evmengine/keeper/helpers.go
- story/client/x/evmengine/keeper/upgrades.go
- story/client/app/cmtlog.go
- story/client/app/sdklog.go
- story/client/app/start.go
- story/client/app/keepers/types.go
- story/client/app/prouter.go
- story/client/app/app.go
- story/client/app/privkey.go
- story/client/app/upgrades/historical.go
- story/client/app/upgrades/types.go
- story/client/app/upgrades/v0_12_1/constants.go
- story/client/app/upgrades/v0_12_1/upgrades.go
- story/client/app/abci.go
- story/client/app/app_config.go
- story/client/app/upgrades.go
- story/client/collections/queue.go
- cosmos-sdk/proto/cosmos/distribution/v1beta1/distribution.proto
- cosmos-sdk/proto/cosmos/distribution/v1beta1/query.proto
- cosmos-sdk/proto/cosmos/distribution/v1beta1/tx.proto
- cosmos-sdk/proto/cosmos/staking/v1beta1/genesis.proto
- cosmos-sdk/proto/cosmos/staking/v1beta1/staking.proto
- cosmos-sdk/proto/cosmos/staking/v1beta1/tx.proto
- cosmos-sdk/simapp/export.go
- cosmos-sdk/x/distribution/abci.go
- cosmos-sdk/x/distribution/client/cli/tx.go
- cosmos-sdk/x/distribution/keeper/allocation.go
- cosmos-sdk/x/distribution/keeper/delegation.go
- cosmos-sdk/x/distribution/keeper/fee_pool.go
- cosmos-sdk/x/distribution/keeper/genesis.go
- cosmos-sdk/x/distribution/keeper/grpc_query.go
- cosmos-sdk/x/distribution/keeper/hooks.go
- cosmos-sdk/x/distribution/keeper/invariants.go
- cosmos-sdk/x/distribution/keeper/keeper.go
- cosmos-sdk/x/distribution/keeper/msg_server.go
- cosmos-sdk/x/distribution/keeper/params.go
- cosmos-sdk/x/distribution/keeper/ubi.go
- cosmos-sdk/x/distribution/keeper/validator.go
- cosmos-sdk/x/distribution/simulation/genesis.go
- cosmos-sdk/x/distribution/simulation/operations.go
- cosmos-sdk/x/distribution/simulation/proposals.go
- cosmos-sdk/x/distribution/types/codec.go

- cosmos-sdk/x/distribution/types/delegator.go
- cosmos-sdk/x/distribution/types/fee_pool.go
- cosmos-sdk/x/distribution/types/msg.go
- cosmos-sdk/x/distribution/types/params.go
- cosmos-sdk/x/distribution/types/params_legacy.go
- cosmos-sdk/x/distribution/types/proposal.go
- cosmos-sdk/x/distribution/types/querier.go
- cosmos-sdk/x/distribution/types/query.pb.gw.go
- cosmos-sdk/x/evidence/go.mod
- cosmos-sdk/x/evidence/go.sum
- cosmos-sdk/x/evidence/keeper/infraction.go
- cosmos-sdk/x/evidence/types/expected_keepers.go
- cosmos-sdk/x/gov/keeper/deposit.go
- cosmos-sdk/x/gov/keeper/keeper.go
- cosmos-sdk/x/gov/module.go
- cosmos-sdk/x/gov/types/expected_keepers.go
- cosmos-sdk/x/slashing/keeper/infractions.go
- cosmos-sdk/x/slashing/simulation/genesis.go
- cosmos-sdk/x/slashing/types/expected_keepers.go
- cosmos-sdk/x/staking/client/cli/flags.go
- cosmos-sdk/x/staking/client/cli/tx.go
- cosmos-sdk/x/staking/client/cli/utils.go
- cosmos-sdk/x/staking/keeper/abci.go
- cosmos-sdk/x/staking/keeper/delegation.go
- cosmos-sdk/x/staking/keeper/genesis.go
- cosmos-sdk/x/staking/keeper/grpc_query.go
- cosmos-sdk/x/staking/keeper/msg_server.go
- cosmos-sdk/x/staking/keeper/msgs.go
- cosmos-sdk/x/staking/keeper/params.go
- cosmos-sdk/x/staking/keeper/period_delegation.go
- cosmos-sdk/x/staking/keeper/slash.go
- cosmos-sdk/x/staking/keeper/val_state_change.go
- cosmos-sdk/x/staking/keeper/validator.go
- cosmos-sdk/x/staking/simulation/genesis.go
- cosmos-sdk/x/staking/simulation/operations.go
- cosmos-sdk/x/staking/types/delegation.go
- cosmos-sdk/x/staking/types/errors.go
- cosmos-sdk/x/staking/types/events.go
- cosmos-sdk/x/staking/types/exported.go
- cosmos-sdk/x/staking/types/genesis.go
- cosmos-sdk/x/staking/types/keys.go
- cosmos-sdk/x/staking/types/msg.go
- cosmos-sdk/x/staking/types/params.go
- cosmos-sdk/x/staking/types/period_delegation.go
- cosmos-sdk/x/staking/types/validator.go

Repository details

- **Repository 1 URL:** https://github.com/piplabs/story-geth
- **Commit hash:** ab8925de20576333b7d8f638d46cc9eab288b4f5

- **Repository 2 URL:** https://github.com/piplabs/story
- **Commit hash:** 20fed5ed45d39c9ac59ab17c03ff3b1efac0f7b2

- **Repository 3 URL:** https://github.com/piplabs/cosmos-sdk
- **Commit hash:** c553dc6734da1d49fcde55efd825d84998051dd7

Note: Issues were reported to the team with intermediate reports during the audit time frame while development was still ongoing. Because of that, some of the issues are already addressed in the hashes above.

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

Lambda is a security researcher and developer with multiple years of experience in IT security and traditional finance. This experience combined with his academic background in Mathematical Finance and Computer Science enables him to thoroughly examine even the most complicated code bases, resulting in several top placements in various audit contests.

Christian Vari is a software engineer with a master's degree in cybersecurity and blockchain technology, certified in Substrate, Cosmos SDK, and CosmWasm development. With experience in designing distributed systems and developing microservices in Golang, Rust, and Haskell, Christian has audited over 100 smart contracts and blockchain systems, including Substrate pallets, Cosmos SDK modules, CosmWasm contracts, Solana programs, and EVM implementations. He has also supported organizations in disaster recovery, investigations and fund rescue.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project kept code as simple as possible, reducing attack risks
Documentation	**Good**	An extensive documentation of the tokenomics was available, but the code documentation could be more extensive.
Best practices	**Good**	Project uses battle-tested libraries and dependencies.
Centralization risks	**Good**	The project does not introduce significant unnecessary centralization risks.

Findings

High severity findings

TRST-H-1 Attackers can perform DoS by spamming unstake and redelegate events

- **Category:** DoS attacks
- **Source:** client/x/evmstaking/keeper/keeper.go
- **Status:** Acknowledged

Description

The Cosmos chain is configured with the **maxGas** parameter set to -1 and lacks an *AnteHandler()* implementation, resulting in the absence of gas metering. Consequently, payloads retrieved from Geth execute all associated events without metering their computational costs. In fact, although fees deducted in associated Solidity contracts are intended to cover these costs, there is no direct correlation between the fees and the actual resource usage on Cosmos.

This discrepancy creates a vulnerability to Denial-of-Service (DoS) attacks and scalability issues, with blocks potentially varying significantly in size and processing time based on the volume of emitted Solidity events.

Operations such as redelegation and unstaking are particularly susceptible to exploitation:

Redelegation

Adding a redelegation to the queue incurs no fee, despite the process being resource-intensive. Additionally, this involves inserting a **Redelegation** object to the queue, with execution deferred to the *EndBlocker()*, which processes all matured redelegations via the *DequeueAllMatureRedelegationQueue()* function.

```
func (k Keeper) DequeueAllMatureRedelegationQueue(ctx context.Context,
currTime time.Time) (matureRedelegations []types.DVVTriplet, err error) {
    store := k.storeService.OpenKVStore(ctx)

    // gets an iterator for all timeslices from time 0 until the current
Blockheader time
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    redelegationTimesliceIterator, err := k.RedelegationQueueIterator(ctx,
sdkCtx.HeaderInfo().Time)
    if err != nil {
        return nil, err
    }
    defer redelegationTimesliceIterator.Close()

    for ; redelegationTimesliceIterator.Valid();
redelegationTimesliceIterator.Next() {
        timeslice := types.DVVTriplets{}
        value := redelegationTimesliceIterator.Value()
```

```
        if err = k.cdc.Unmarshal(value, &timeslice); err != nil {
            return nil, err
        }

        matureRedelegations = append(matureRedelegations,
timeslice.Triplets...)

        if err = store.Delete(redelegationTimesliceIterator.Key()); err !=
nil {
            return nil, err
        }
    }

    return matureRedelegations, nil
}
```

An attacker can spam the queue with numerous redelegations targeting the same time slice, overloading the system when the timer is triggered and potentially leading to consensus timeouts.

Unstake

Unstaking operations, particularly from non-existent delegations or validators, generate **Withdraw** events without incurring fees. Those events are then processed in the *ProcessWithdraw()* function until erroring in the following line of the Cosmos SDK staking module:

```
    // check validator existence
    if _, err := k.GetValidator(ctx, validatorAddress); err != nil {
        return err
    }
```

This allows attackers to flood the network with such events, triggering unnecessary logic execution and amplifying the risk of a DoS attack.

The absence of gas metering exacerbates scalability challenges by causing block sizes to vary unpredictably based on the number of emitted Solidity events, which leads to processing delays. Additionally, larger blocks prolong the time required for peer-to-peer (P2P) gossip between nodes.

Recommended mitigation

As a short-term mitigation, it is recommended to charge a static fee within the Solidity smart contract for operations such as redelegation and unstaking, ensuring some level of cost alignment with the computational resources consumed on the Cosmos chain.

However, for a robust and long-term solution, it is recommended to implement the following measures:

- Gas metering should be introduced on the Cosmos chain by configuring an appropriate **maxGas** value for blocks and deploying an *AnteHandler()*. This will ensure that all operations are metered accurately based on their computational cost, preventing unmetered payloads from executing unchecked. It is not necessary to actually charge for gas, this has the only purpose of limiting the number of events that can fit the block.
- Fees must be aligned with actual resource consumption. All operations executed on Cosmos should require a fee proportional to their computational load, preventing the network from being overwhelmed by large, unmetered event payloads.
- Monitoring the network and enforcing block size limits will ensure predictable processing times and mitigate the risk of large blocks delaying peer-to-peer (P2P) communication and consensus.

Team response

The team acknowledges the issue and states that they don't allow cosmos side tx and users don't have cosmos side balance. Tx and gas metering already happen on evm side. The current plan is to add additional fee on gas on our evm side staking contract as the gas meter.

Additionally, although it's a flat fee right now, it can be adjusted to an event specific pricing later on depending on how resource intensive each call is.

TRST-H-2 IP Graph precompile approximation of required gas can significantly underestimate gas usage

- **Category:** DoS attacks
- **Source:** core/vm/ipgraph.go
- **Status:** Fixed

Description

The *RequiredGas()* function of the IP Graph precompiles returns the fixed value **ipGraphWriteGas** for the *addParentIp()* function:

```
func (c *ipGraph) RequiredGas(input []byte) uint64 {
  // Smart contract function's selector is the first 4 bytes of the input
  if len(input) < 4 {
    return intrinsicGas
  }

  selector := input[:4]

  switch {
  case bytes.Equal(selector, addParentIpSelector):
    return ipGraphWriteGas
```

In Geth, this function is used to determine the gas that is charged on execution, and it should therefore be close to the effective resources that are used by a function call, as the gas accounting mechanism does not work as intended otherwise. The problem is that it is possible to add many parent IPs with one call to *addParentIP()* and the function iterates over all of

them and performs storage writes for them. The gas usage is therefore roughly linear in the number of parent IPs times the write gas. In theory, Story's smart contracts (which are the only instance that is allowed to perform write operations) limit the maximum parents to 8, which would limit the impact of this underestimation (it would be off by a factor of 8 in the worst case). However, this check is not effective here because it is only performed after the precompile was called. Concretely, the licensing module calls *registerDerivativeIP()* without restricting the length of **parentIpIds**.

```
    function registerDerivative(
        address childIpId,
        address[] calldata parentIpIds,
        uint256[] calldata licenseTermsIds,
        address licenseTemplate,
        bytes calldata royaltyContext,
        uint256 maxMintingFee
    ) external whenNotPaused nonReentrant verifyPermission(childIpId) {
        if (parentIpIds.length != licenseTermsIds.length) {
            revert
Errors.LicensingModule__LicenseTermsLengthMismatch(parentIpIds.length,
licenseTermsIds.length);
        }
        if (parentIpIds.length == 0) {
            revert Errors.LicensingModule__NoParentIp();
        }

        // ...

        LICENSE_REGISTRY.registerDerivativeIp(childIpId, parentIpIds,
licenseTemplate, licenseTermsIds, false);

        // ...

        ROYALTY_MODULE.onLinkToParents(childIpId, parentIpIds,
royaltyPolicies, royaltyPercents, royaltyContext);
```

Later, *registerDerivativeIP()* calls the precompile:

```
    function registerDerivativeIp(
        address childIpId,
        address[] calldata parentIpIds,
        address licenseTemplate,
        uint256[] calldata licenseTermsIds,
        bool isUsingLicenseToken
    ) external onlyLicensingModule {
        LicenseRegistryStorage storage $ = _getLicenseRegistryStorage();
        if (_isDerivativeIp(childIpId)) {
            revert
Errors.LicenseRegistry__DerivativeAlreadyRegistered(childIpId);
        }
```

```
        if ($.childIps[childIpId].length() > 0) {
            revert
Errors.LicenseRegistry__DerivativeIpAlreadyHasChild(childIpId);
        }
        if ($.attachedLicenseTerms[childIpId].length() > 0) {
            revert
Errors.LicenseRegistry__DerivativeIpAlreadyHasLicense(childIpId);
        }
        // ...

        IP_GRAPH_ACL.allow();
        (bool success, ) = IP_GRAPH.call(
            abi.encodeWithSignature("addParentIp(address,address[])",
childIpId, parentIpIds)
        );
        IP_GRAPH_ACL.disallow();
```

Only in the subsequent call to *onLinkToParents()*, the length is checked:

```
    function onLinkToParents(
        address ipId,
        address[] calldata parentIpIds,
        address[] calldata licenseRoyaltyPolicies,
        uint32[] calldata licensesPercent,
        bytes calldata externalData
    ) external nonReentrant onlyLicensingModule {
        RoyaltyModuleStorage storage $ = _getRoyaltyModuleStorage();

        // If an IP already has a vault, it means that it's either a root
node which cannot link to parents
        // or it's a derivative in which case it cannot link to parents
either
        if ($.ipRoyaltyVaults[ipId] != address(0)) revert
Errors.RoyaltyModule__UnlinkableToParents();

        if (parentIpIds.length == 0) revert
Errors.RoyaltyModule__NoParentsOnLinking();
        if (parentIpIds.length > $.maxParents) revert
Errors.RoyaltyModule__AboveParentLimit();
        if (_getAncestorCount(ipId) > $.maxAncestors) revert
Errors.RoyaltyModule__AboveAncestorsLimit();
```

An attacker can abuse this by creating calls to *registerDerivative()* with a huge number of entries in the **parentIpIds** array. They will eventually revert (in *onLinkToParents()*), but the precompile is executed before that and has to iterate over all of the provided addresses, while the attacker is only paying very little gas fees for the precompile call. If there are multiple such calls in a block, work that the nodes must perform is no longer effectively bounded by the block gas limit, because the charged gas for these calls is inaccurate. In the worst case, this

could lead to DoS situations where nodes cannot produce the next block in time and the chain does not progress.

Recommended mitigation

Parse the number of entries in the array when calculating the required gas and multiply **ipGraphWriteGas** by it.

Team response

Fixed in PR #63.

Mitigation review

Fixed. The recommended mitigation was implemented.

TRST-H-3 Loss of rewards in case of multiple delegations

- **Category:** Logical flaws
- **Source:** x/distribution/keeper/hooks.go
- **Status:** Fixed

Description

When a user stakes multiple times through the Solidity contract, the Cosmos SDK's staking module automatically redeems rewards to the user's Cosmos address.

Specifically, the user initiates staking by calling the *stake()* function on the Solidity contract which emits a **Deposit** event.

```
   function _stake(
        bytes calldata delegatorUncmpPubkey,
        bytes calldata validatorUncmpPubkey,
        IIPTokenStaking.StakingPeriod stakingPeriod,
        bytes calldata data
    ) internal returns (uint256) {
        // This can't be tested from Foundry (Solidity), but can be
triggered from js/rpc
        require(stakingPeriod <= IIPTokenStaking.StakingPeriod.LONG,
"IPTokenStaking: Invalid staking period");
        (uint256 stakeAmount, uint256 remainder) =
roundedStakeAmount(msg.value);
        require(stakeAmount >= minStakeAmount, "IPTokenStaking: Stake amount
under min");

        uint256 delegationId = 0;
        if (stakingPeriod != IIPTokenStaking.StakingPeriod.FLEXIBLE) {
            delegationId = ++_delegationIdCounter;
        }
        emit Deposit(
            delegatorUncmpPubkey,
            validatorUncmpPubkey,
            stakeAmount,
```

```
            uint8(stakingPeriod),
            delegationId,
            msg.sender,
            data
        );
```

The **Deposit** event is captured in the evmstaking module in the *ProcessStakingEvents()* function and the *ProcessDeposit()* function is called.

```
        case types.DepositEvent.ID:

            ev, err := k.ParseDepositLog(ethlog)
            if err != nil {
                clog.Error(ctx, "Failed to parse Deposit log", err)
                continue
            }
            ev.StakeAmount.Div(ev.StakeAmount, gwei)
            if err = k.ProcessDeposit(ctx, ev); err != nil {
                clog.Error(ctx, "Failed to process deposit", err)
                continue
            }
```

The *ProcessDeposit()* function executes its logic, constructs a **MsgDelegate** and then invokes the *Delegate()* function of the staking keeper.

```
    msg := stypes.NewMsgDelegate(
        depositorAddr.String(), validatorAddr.String(), amountCoin,
        delID, periodType,
    )
    _, err = skeeperMsgServer.Delegate(ctx, msg)
```

Within the *Delegate()* function, the *BeforeDelegationSharesModified()* hook is called in case a **Delegation** is already registered for the target delegator, triggering the respective hook in the staking module.

```
    delegation, err := k.GetDelegation(ctx, delAddr, valbz)
    if err == nil {
        // found
        err = k.Hooks().BeforeDelegationSharesModified(ctx, delAddr, valbz)
```

The hook executes the *withdrawDelegationRewards()* function, which distributes rewards directly to the delegator's Cosmos address.

```
func (h Hooks) BeforeDelegationSharesModified(ctx context.Context, delAddr
sdk.AccAddress, valAddr sdk.ValAddress) error {
    val, err := h.k.stakingKeeper.Validator(ctx, valAddr)
    if err != nil {
        return err
```

```
    }

    del, err := h.k.stakingKeeper.Delegation(ctx, delAddr, valAddr)
    if err != nil {
        return err
    }

    if _, err := h.k.withdrawDelegationRewards(ctx, val, del); err != nil {
        return err
    }
```

This bypasses the defined withdrawal queue, resulting in those funds to be stuck in the delegator's Cosmos address.

Recommended mitigation

It is recommended to distribute rewards accumulated through repeated delegations in the EVM through the withdrawal queue.

Team response

Fixed.

Mitigation review

Fixed. This case is now correctly handled during reward withdrawal.

TRST-H-4 Order of EVM Events is not respected

- **Category:** Logical flaws
- **Source:** client/x/evmstaking/keeper/keeper.go
- **Status:** Fixed

Description

In keeper.go, there is the function *ProcessStakingEvents()* that iterates over all events which are emitted from the staking event (in one block) and processes them depending on the log topic:

```
func (k Keeper) ProcessStakingEvents(ctx context.Context, height uint64,
logs []*evmenginetypes.EVMEvent) error {
  gwei, exp := big.NewInt(10), big.NewInt(9)
  gwei.Exp(gwei, exp, nil)

  for _, evmLog := range logs {
    if err := evmLog.Verify(); err != nil {
      return errors.Wrap(err, "verify log [BUG]") // This shouldn't happen
    }
    ethlog, err := evmLog.ToEthLog()
    if err != nil {
      return err
    }
```

```
    // TODO: handle when each event processing fails.

    // Convert the amount from wei to gwei (Eth2 spec withdrawal is
specified in gwei) by dividing by 10^9.
    // TODO: consider rounding and decimal precision when dividing bigint.

    switch ethlog.Topics[0] {
```

However, this iteration is not performed in the order of emission for these events. The following code enforces a particular order for prevPayloadEvents:

```
  // Collect local view of the evm logs from the previous payload.
  evmEvents, err := s.evmEvents(ctx, payload.ParentHash)
  if err != nil {
    return nil, errors.Wrap(err, "prepare evm event logs")
  }

  // Ensure the proposed evm event logs are equal to the local view.
  if err := evmEventsEqual(evmEvents, msg.PrevPayloadEvents); err != nil {
    return nil, errors.Wrap(err, "verify prev payload events")
  }
```

They need to have the same order as the function evmEvents returns, which is sorted by Address > Topics > Data:

```
    // Sort by Address > Topics > Data
    // This avoids dependency on runtime ordering.
    sort.Slice(events, func(i, j int) bool {
        if cmp := bytes.Compare(events[i].Address, events[j].Address); cmp
!= 0 {
            return cmp < 0
        }

        topicI := slices.Concat(events[i].Topics...)
        topicJ := slices.Concat(events[j].Topics...)
        if cmp := bytes.Compare(topicI, topicJ); cmp != 0 {
            return cmp < 0
        }

        return bytes.Compare(events[i].Data, events[j].Data) < 0
    })
```

This is problematic because the order is important for some calls. For instance, if a user submits three calls *addOperator(A)*, *removeOperator(A)*, *addOperator(B)* (in this order), they expect a result of operator B being added. If they are in the same block, this is not guaranteed, and data corruption can happen based on the order of their topics and then the order of the data. Most likely, all *addOperator()* or *removeOperator()* calls would be processed first and then the other ones

Recommended mitigation

Enforce that the logs are processed according to their log index.

Team response

Fixed in PR #325.

Mitigation review

Fixed. The logs are now processed in order.

TRST-H-5 If validator's moniker string length is above a certain threshold, it would lead to lost stake

- **Category:** Logical flaws
- **Source:** client/x/evmstaking/keeper/validator.go
- **Status:** Fixed

Description

When creating a validator, the user can provide a Moniker in the smart contract that is emitted in the event. When processing this event, validator.go passes it to the MsgCreateValidator without any further processing (if it is non-empty, otherwise the validator address is used):

```
  moniker := ev.Moniker
  if moniker == "" {
    moniker = validatorAddr.String() // use validator address as moniker if
not provided
  }

  minSelfDelegation, err := k.stakingKeeper.MinDelegation(ctx)
  if err != nil {
    return errors.Wrap(err, "get min self delegation")
  }

  // Validator does not exist, create validator with self-delegation.
  msg, err := stypes.NewMsgCreateValidator(
    validatorAddr.String(),
    validatorPubkey,
    amountCoin,
    stypes.Description{Moniker: moniker},
    stypes.NewCommissionRates(
      // Divide these decimals by 100 to convert from basis points to
decimal. Will cut off decimal as the rates are integers.

math.LegacyNewDec(int64(ev.CommissionRate)).Quo(math.LegacyNewDec(10000)),

math.LegacyNewDec(int64(ev.MaxCommissionRate)).Quo(math.LegacyNewDec(10000))
,
```

```
math.LegacyNewDec(int64(ev.MaxCommissionChangeRate)).Quo(math.LegacyNewDec(1
0000)),
    ),
    minSelfDelegation, // make minimum self delegation align with minimum
delegation amount
    int32(ev.SupportsUnlocked),
  )
```

However, the cosmos-sdk code calls *EnsureLength()* on the description within the msg validation logic:

```
func (k msgServer) ValidateCreateValidatorMsg(ctx context.Context, msg
*types.MsgCreateValidator) error {
  // ...
  if _, err := msg.Description.EnsureLength(); err != nil {
    return err
  }
```

The *EnsureLength()* function enforces a maximum length (of 70 characters) for the Moniker:

```
// EnsureLength ensures the length of a validator's description.
func (d Description) EnsureLength() (Description, error) {
  if len(d.Moniker) > MaxMonikerLength {
    return d, errors.Wrapf(sdkerrors.ErrInvalidRequest, "invalid moniker
length; got: %d, max: %d", len(d.Moniker), MaxMonikerLength)
  }
```

Therefore, if the user specifies a longer moniker, the creation of the validator will fail and the initial stake will be lost.

Recommended mitigation

Restrict the moniker to 70 characters in the smart contract.

Team response

Fixed in PR #360.

Mitigation review

Fixed. The moniker length is now restricted.

Medium severity findings

TRST-M-1 Discrepancy between operator management in smart contract and L1

- **Category:** Logical flaws
- **Source:** client/x/evmstaking/keeper/delegator_address.go
- **Status:** Fixed

Description

The smart contracts contain a function *removeOperator()* that removes a specified operator:

```
    /// @notice Removes an operator for a delegator.
    /// @param uncmpPubkey 65 bytes uncompressed secp256k1 public key.
    /// @param operator The operator address to remove.
    function removeOperator(
        bytes calldata uncmpPubkey,
        address operator
    ) external verifyUncmpPubkeyWithExpectedAddress(uncmpPubkey, msg.sender)
{
        emit RemoveOperator(uncmpPubkey, operator);
    }
```

However, the **operator** argument of the emitted event is completely ignored when this event is processed:

```
func (k Keeper) ProcessRemoveOperator(ctx context.Context, ev
*bindings.IPTokenStakingRemoveOperator) (err error) {
  defer func() {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    if err != nil {
      sdkCtx.EventManager().EmitEvents(sdk.Events{
        sdk.NewEvent(
          types.EventTypeRemoveOperatorFailure,
          sdk.NewAttribute(types.AttributeKeyBlockHeight,
strconv.FormatInt(sdkCtx.BlockHeight(), 10)),
          sdk.NewAttribute(types.AttributeKeyDelegatorUncmpPubKey,
hex.EncodeToString(ev.UncmpPubkey)),
          sdk.NewAttribute(types.AttributeKeyOperatorAddress,
ev.Operator.Hex()),
          sdk.NewAttribute(types.AttributeKeyStatusCode,
errors.UnwrapErrCode(err).String()),
        ),
      })
    }
  }()

  delCmpPubkey, err := UncmpPubKeyToCmpPubKey(ev.UncmpPubkey)
  if err != nil {
    return errors.WrapErrWithCode(errors.InvalidUncmpPubKey,
errors.Wrap(err, "compress delegator pubkey"))
```

```
  }
  depositorPubkey, err := k1util.PubKeyBytesToCosmos(delCmpPubkey)
  if err != nil {
    return errors.Wrap(err, "depositor pubkey to cosmos")
  }

  depositorAddr := sdk.AccAddress(depositorPubkey.Address().Bytes())

  if err := k.DelegatorOperatorAddress.Remove(ctx, depositorAddr.String());
err != nil {
    return errors.Wrap(err, "delegator operator address map remove")
  }

  return nil
}
```

The code will just remove the current operator.

A similar discrepancy exists for *addOperator()*. The smart contract mentions that an operator is added (which makes sense based on the function name). However, in the L1 code, there can only be ever one operator per delegator (stored in the **k.DelegatorOperatorAddress** map) and the current operator is overwritten whenever a new one is added.

Recommended mitigation

If multiple operators should be supported, we recommend changing the L1 code and data structures to store all operators. If not, we recommend changing the smart contracts (function name and documentation for *addOperator()*, removal of unnecessary parameter for *removeOperator()*).

Team response

Fixed.

Mitigation review

Fixed. The client resolved the issue by modifying the Solidity smart contracts and the corresponding handlers to enhance clarity regarding the functionality's usage. Specifically, functions were renamed to **setOperator** and **unsetOperator** to better reflect their purpose.

TRST-M-2 Vulnerability in CometBFT dependency may allow system compromise

- **Category:** Dependency issues
- **Source:** go.mod
- **Status:** Fixed

Description

The Cosmos chain relies on **cometbft@v0.38.9**, a version impacted by an high severity vulnerability outlined in GHSA-g5xx-c4hv-9ccc.

This issue allows a malicious actor to exploit the system by providing a compromised snapshot. Nodes joining the network and syncing with this snapshot may inadvertently trigger chain splits, disrupting network consensus and fragmenting the blockchain.

Recommended mitigation

It is recommended to upgrade to a patched version of CometBFT that resolves this vulnerability to safeguard the network's integrity and consensus stability.

Team response

Fixed.

Mitigation review

Fixed. The client resolved the issue by updating the affected dependencies to the suggested versions.

TRST-M-3 Lack of transaction atomicity in payload execution

- **Category:** Logical flaws
- **Source:** client/x/evmengine/msg_server.go
- **Status:** Fixed

Description

In the Cosmos SDK, transactions are designed to be atomic, meaning that all operations within a transaction either succeed entirely and commit to the blockchain or fail completely, reverting any pending changes.

However, the *MsgExecutionPayload()* handler processes sub-transactions derived from Geth payload events as a single transaction without enforcing atomicity. This absence of rollback functionality introduces a risk of partial or inconsistent state changes when sub-transactions fail.

This issue extends to all messages processed by the *MsgExecutionPayload()* handler. Events triggering critical Cosmos SDK message handlers, such as *Delegate()*, *Undelegate()*, and *CreateValidator()*, which are designed to revert entire transactions upon encountering an error, fail to uphold atomicity due to the current implementation.

For instance:

- A validator could end up with zero voting power (a state Cosmos SDK typically prevents) if self-delegation fails during *CreateValidator()*, yet the validator is still created.
- During undelegation, an error from exceeding the active undelegation cap may lead to incorrectly updated state, as the rollback does not occur.
- The *stake()* function in the associated Solidity contract burns a user's tokens and triggers delegation on the Cosmos chain by emitting a **Deposit** event. The **MsgExecutionPayload** handler processes this event through the *ProcessStakingEvents()* function, specifically by *ProcessDeposit()* for **Deposit** events.

This function mints coins within the module and transfers them to the delegator's address. However, if a user attempts to delegate to a non-existent validator, as seen an error occurs.

```
    delID := ev.DelegationId.String()
    periodType := int32(ev.StakingPeriod.Int64())

    val, err := k.stakingKeeper.GetValidator(ctx, validatorAddr)
    if errors.Is(err, stypes.ErrNoValidatorFound) {
        return errors.WrapErrWithCode(errors.ValidatorNotFound,
errors.New("validator not exists"))
    } else if err != nil {
        return errors.Wrap(err, "get validator failed")
    }
```

Despite this failure, prior state changes, such as minting and transferring coins, are not reverted. This results in minted coins remaining with the delegator without corresponding delegation, creating an inconsistent state.

Recommended mitigation

As a short-term mitigation, it is recommended to wrap the execution of each event within a **CachedCtx**. This approach ensures that any changes made during the event's execution are rolled back in case of a failure but would slightly degrade performances.

For a more efficient and long-term solution, it is recommended to modify the *PrepareProposal()* method of the ABCI++ to split events in the payload into multiple transactions when constructing the block. By isolating each event as a separate transaction, this solution ensures atomicity across the entire payload and aligns with the Cosmos SDK's transactional guarantees.

Team response

Fixed.

Mitigation review

Fixed. The client addressed the issue by implementing **CachedContext** for each event handler. This approach allows the execution to use a cached context that is written to the store only upon successful execution. This resolves the problem of partial states being written in the event of errors or panics, ensuring state consistency.

TRST-M-4 Unenforced determinism and unpredictable duration of ABCI++ events could lead to consensus issue and affect chain liveliness and throughput

- **Category:** DoS attacks
- **Source:** client/x/evmengine/keeper/abci.go
- **Status:** Acknowledged

Description

In the *ExecutionPayload()* message handler, asynchronous calls to a Geth node are made during transaction processing, introducing variability due to network and infrastructure conditions.

```
func (s msgServer) ExecutionPayload(ctx context.Context, msg
*types.MsgExecutionPayload) (*types.ExecutionPayloadResponse, error) {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    if sdkCtx.ExecMode() != sdk.ExecModeFinalize {
        return nil, errors.New("only allowed in finalize mode")
    }

    payload, err := s.parseAndVerifyProposedPayload(ctx, msg)
    if err != nil {
        return nil, err
    }

    // TODO: should we compare and reject in a finalized block?
    //// Ensure that the withdrawals in the payload are from the front
indices of the queue.
    // if err := s.compareWithdrawals(ctx, payload.Withdrawals); err != nil
{
    //  return nil, errors.Wrap(err, "compare local and received
withdrawals")
    //}

    // TODO: We dequeue with assumption that the top items of the queue are
the ones that are processed in the block.
    // TODO: We might need to check that the withdrawals in the finalized
block are the same as the ones dequeued.
    // Since we already checked the withdrawals in the proposal server, we
simply check the length here.
    log.Debug(
        ctx, "Dequeueing eligible withdrawals [BEFORE]",
        "total_len", len(payload.Withdrawals),
    )
    maxWithdrawals, err := s.evmstakingKeeper.MaxWithdrawalPerBlock(ctx)
    if err != nil {
        return nil, errors.Wrap(err, "error getting max withdrawal per
block")
    }
    ws, err := s.evmstakingKeeper.DequeueEligibleWithdrawals(ctx,
maxWithdrawals)
    if err != nil {
        return nil, errors.Wrap(err, "error on withdrawals dequeue")
    }
    log.Debug(
        ctx, "Dequeueing eligible withdrawals [AFTER]",
        "total_len", len(payload.Withdrawals),
        "withdrawals_len", len(ws),
```

```
    )

    if len(ws) > len(payload.Withdrawals) {
        return nil, fmt.Errorf(
            "dequeued withdrawals %v should not greater than proposed
withdrawals %v",
            len(ws), len(payload.Withdrawals),
        )
    }

    log.Debug(
        ctx, "Dequeueing eligible reward withdrawals [BEFORE]",
        "total_len", len(payload.Withdrawals),
        "withdrawals_len", len(ws),
    )
    maxRewardWithdrawals := maxWithdrawals - uint32(len(ws))
    rws, err := s.evmstakingKeeper.DequeueEligibleRewardWithdrawals(ctx,
maxRewardWithdrawals)
    if err != nil {
        return nil, errors.Wrap(err, "error on reward withdrawals dequeue")
    }
    log.Debug(
        ctx, "Dequeueing eligible reward withdrawals [AFTER]",
        "total_len", len(payload.Withdrawals),
        "withdrawals_len", len(ws),
        "reward_withdrawals_len", len(rws),
    )

    if totalWithdrawals := len(ws) + len(rws); totalWithdrawals !=
len(payload.Withdrawals) {
        return nil, fmt.Errorf(
            "dequeued total withdrawals %v should equal to proposed
withdrawals %v",
            totalWithdrawals, len(payload.Withdrawals),
        )
    }

    err = retryForever(ctx, func(ctx context.Context) (bool, error) {
        status, err := pushPayload(ctx, s.engineCl, payload)
        if err != nil || isUnknown(status) {
            // We need to retry forever on networking errors, but can't
easily identify them, so retry all errors.
            log.Warn(ctx, "Processing finalized payload failed: push new
payload to evm (will retry)", err,
                "status", status.Status)

            return false, nil // Retry
        } else if invalid, err := isInvalid(status); invalid {
            // This should never happen. This node will stall now.
            log.Error(ctx, "Processing finalized payload failed; payload
invalid [BUG]", err)
```

```
            return false, err // Don't retry, error out.
        } else if isSyncing(status) {
            log.Warn(ctx, "Processing finalized payload; evm syncing", nil)
        }

        return true, nil // We are done, don't retry
    })
    if err != nil {
        return nil, err
    }
```

Since ABCI++ applications must function as deterministic finite-state machines to ensure secure replication by the CometBFT consensus engine, such non-determinism violates core principles. Divergent responses from external calls may lead nodes to produce inconsistent blocks, potentially causing chain halts.

Additionally, both *ExecutionPayload()* and *PrepareProposal()* functions employ the *retryForever()* function to retry calls with a backoff strategy (as well as a **buildDelay** timeout) to handle Geth requests. This design risks delays during consensus rounds if processing times exceed the CometBFT consensus timeout. Consequently, variability in block creation times may lead to slowdowns or even chain halts if proposers fail to retrieve payloads within the timeout window.

```
    waitTo := triggeredAt.Add(k.buildDelay)
    select {
    case <-ctx.Done():
        return nil, errors.Wrap(ctx.Err(), "context done")
    case <-time.After(time.Until(waitTo)):
    }

    // Fetch the payload (retrying on network errors).
    var payloadResp *engine.ExecutionPayloadEnvelope
    err = retryForever(ctx, func(ctx context.Context) (bool, error) {
        var err error
        payloadResp, err = k.engineCl.GetPayloadV3(ctx, payloadID)
        if isUnknownPayload(err) {
            return false, err
        } else if err != nil {
            log.Warn(ctx, "Preparing proposal failed: get evm payload (will
retry)", err)
            return false, nil
        }

        return true, nil
    })
    if err != nil {
        return nil, err
    }
```

While those issues are less likely in controlled testnet environments with skilled validator operators and optimized infrastructure, on a mainnet where anyone can operate a validator, the risk is exacerbated by the presence of misconfigured or underperforming nodes, increasing the likelihood of instability during periods of heavy network load.

Recommended mitigation

It is recommended to implement the following measures:

- **Eliminate asynchronous external calls in ABCI++ methods**: The *PrepareProposal()* function and the **ExecutionPayload** message handler should avoid making asynchronous calls to external nodes such as Geth. Application state changes must derive solely from deterministic block execution events triggered by CometBFT to ensure consensus integrity and correctness.
- **Decouple payload retrieval**: External payload retrieval should be moved outside the ABCI++ consensus operations. It should be moved to a side thread which directly provides the required data in the mempool. By decoupling this process, the risk of consensus delays due to infrastructure variability or network latency can be mitigated.
- **Refactor retry logic**: Replace **buildDelay** and *retryForever()* with a bounded retry strategy that respects the CometBFT consensus timeout window.
- **Leverage of network monitoring tools**: Implement comprehensive monitoring tools to track key metrics such as block time, node health, and liveliness. These tools can provide real-time insights into the network's operational status, enabling early detection of performance bottlenecks or misconfigured nodes.

Team response

The client acknowledges the issue and states that the engineAPI call flow is largely consistent with the Ethereum architecture. While it is true that misconfigured or underperforming nodes can reduce network performance, this is a common challenge for any network, even in the absence of engineAPI calls.

Additionally, the client has implemented an improvement to address this concern, as detailed in this pull request.

TRST-M-5 Unhandled panics in event execution cause persistent empty block production

- **Category:** DoS attacks
- **Source:** client/x/evmstaking/keeper/keeper.go
- **Status:** Fixed

Description

The *ProcessStakingEvents()* function processes staking events for each block by invoking associated handlers. Deferred functions are implemented in the execution of handlers to emit events in case of failures.

```
    defer func() {
        sdkCtx := sdk.UnwrapSDKContext(ctx)
        if err != nil {
            sdkCtx.EventManager().EmitEvents(sdk.Events{
                sdk.NewEvent(
                    types.EventTypeUpgradeFailure,
                    sdk.NewAttribute(types.AttributeKeyBlockHeight,
strconv.FormatInt(sdkCtx.BlockHeight(), 10)),
                    sdk.NewAttribute(types.AttributeKeyUpgradeName,
ev.Name),
                    sdk.NewAttribute(types.AttributeKeyUpgradeHeight,
strconv.FormatInt(ev.Height, 10)),
                    sdk.NewAttribute(types.AttributeKeyUpgradeInfo,
ev.Info),
                    sdk.NewAttribute(types.AttributeKeyStatusCode,
errors.UnwrapErrCode(err).String()),
                ),
            })
        }
    }()
```

However, this mechanism does not sufficiently address potential panics originating from Cosmos SDK's staking module.

While this is not an issue in standard Cosmos implementation, in this case, since transaction atomicity is not enforced it will cause the execution of the *ExecutePayload()* function to be completely reverted, resulting in the production of empty blocks. The chain will then repeatedly attempts to process the same events, creating a loop of failed executions without any recovery mechanism.

Additionally, no events are emitted to inform node operators about the transaction failures, leaving them without guidance for corrective actions.

Recommended mitigation

It is recommended to implement a recovery mechanism for panics within the *ProcessStakingEvents()* function. This can be achieved by utilizing a recover function to isolate panics for individual events, allowing the system to continue processing other events without a global failure.

Team response

Fixed.

Mitigation review

Fixed. The client resolved the issue by implementing panic recovery logic within deferred functions for each event processing handler.

TRST-M-6 Misconfigurations in HTTP server enables Slowloris and header overload attacks

- **Category:** DoS attacks
- **Source:** client/server/server.go
- **Status:** Fixed

Description

The HTTP server is configured to handle API requests allowing users to perform queries to various modules. However, server security hardening is not enforced.

```
s.httpServer = &http.Server{
    Addr:              listenAddress,
    Handler:           svrHandler,
    ReadHeaderTimeout: 60 * time.Second,
}
```

The **ReadHeaderTimeout** is defined but its duration of one minute is excessively long, leaving the server vulnerable to slow HTTP request attacks, such as a Slowloris attack.

Moreover, other timeouts, namely, **ReadTimeout**, **WriteTimeout**, and **IdleTimeout**, are not configured, exacerbating the risk.

The server also retains the default value of 1 MB for **MaxHeaderBytes**, which can be exploited by malicious actors to overload the server by sending large header payloads.

Recommended mitigation

It is recommended to:

- Configure the **ReadTimeout**, **WriteTimeout**, and **IdleTimeout** settings to reasonable durations, such as 5-10 seconds, depending on the application's expected workload.
- Reduce the **ReadHeaderTimeout** to an appropriate value, such as 10-15 seconds.
- Set **MaxHeaderBytes** to a stricter limit based on the expected size of request headers (e.g., 8 KB).

Team response

Fixed and improved.

Mitigation review

Fixed. The client addressed the issue by implementing **ReadTimeout**, **WriteTimeout**, and **IdleTimeout** timeouts with reasonable durations and setting **MaxHeaderBytes** to an appropriate value.

These parameters are configurable, allowing adjustments as needed.

Low severity findings

TRST-L-1 Inconsistent input length check in IP graph precompiles

- **Category:** Validation flaws
- **Source:** core/vm/ipgraph.go
- **Status:** Fixed

Description

All the precompile functions in ipgraph.go check the input length before accessing any data:

```
func (c *ipGraph) addParentIp(input []byte, evm *EVM, ipGraphAddress
common.Address) ([]byte, error) {
  allowed, err := c.isAllowed(evm)

  if err != nil {
    return nil, err
  }

  if !allowed {
    return nil, fmt.Errorf("caller not allowed to add parent IP")
  }

  log.Info("addParentIp", "input", input)
  if len(input) < 96 {
    return nil, fmt.Errorf("input too short for addParentIp")
  }
```

However, the checked lengths do not always correspond to the actual expected length:

- *setRoyalty()*: Checks for 96 bytes, but should be 128 bytes according to the ABI.
- *getRoyalty()*: Checks for 64 bytes, but should be 96 bytes according to the ABI.
- *getRoyaltyStack()*: Checks for 32 bytes, but should be 64 bytes according to the ABI.

Moreover, it is only checked that the issue is not too short, the caller can append arbitrary bytes to the calldata, which will still be accepted. While this data will be ignored by the functions, it is still recommended to check for the exact length to avoid any integration issues where callers pass in excessive or incorrect data by mistake.

Recommended mitigation

We recommend changing the length checks to be in sync with the ABI definition and enforcing strict equality checks for the input length.

Team response

Fixed.

Mitigation review

Fixed. The client resolved the issue by implementing stricter equality checks.

TRST-L-2 RANDAO implementation can lead to issues in consumers

- **Category:** Entropy issues
- **Source:** client/x/evmengine/abci.go
- **Status:** Acknowledged

Description

In abci.go, the field **Random** in **engine.PayloadAttributes** (which is used as the value for RANDAO) is set to the block hash of the previous block:

```
attrs := &engine.PayloadAttributes{
  Timestamp:             ts,
  Random:                head.Hash(), // We use head block hash as randao.
  SuggestedFeeRecipient: feeRecipient,
  Withdrawals:           withdrawals,
  BeaconRoot:            &appHash,
}
```

This changes the semantics (and safety assumptions) of RANDAO and therefore **block.prevrandao**. With this implementation, the RANDAO value of the next block will always be known. A consequence of this is that some commit-reveal schemes that are used by smart contracts will not be safe. Under some assumptions, it is safe on Ethereum to commit at block X and then use **block.prevrandao** at block X + 2 as "randomness" (e.g. to influence a mint). Because no one (not even validators) will know the RANDAO value of block X + 1, they cannot choose the block X to submit the tx based on this. However, this would not be safe on Story: As the block hash of block X is the RANDAO value of block X + 1, a validator can choose block X such that the resulting **block.prevrandao** value in block X + 2 is favorable for their use case.

Recommended mitigation

We recommend implementing an unpredictable value for RANDAO. Otherwise, the current implementation should at least be clearly documented such that developers do not have wrong assumptions when deploying their contracts on Story.

Team response

The client acknowledges the issue and states that they will document the differences from the standard RANDAO and recommend against its use.

The client further notes that this decision may be revisited, with the possibility of properly implementing RANDAO in the future.

TRST-L-3 Event log filtering in ABCI++ methods could lead to slowdown of block production

- **Category:** DoS attacks
- **Source:** client/x/evmengine/abci.go
- **Status:** Acknowledged

Description

The *PrepareProposal()* function calls the *evmEvents()* method to retrieve event logs from Geth, filtering by the current **blockHash** and targeting IP contracts. This process involves executing *FilterLogs()*, which scans the entire block to identify events matching the specified criteria.

```
func (k *Keeper) evmEvents(ctx context.Context, blockHash common.Hash)
([]*types.EVMEvent, error) {
    var logs []ethtypes.Log
    err := retryForever(ctx, func(ctx context.Context) (fetched bool, err
error) {
        logs, err = k.engineCl.FilterLogs(ctx, ethereum.FilterQuery{
            BlockHash: &blockHash,
            Addresses: []common.Address{
                common.HexToAddress(predeploys.IPTokenStaking),
                common.HexToAddress(predeploys.UBIPool),
                common.HexToAddress(predeploys.UpgradeEntrypoint),
            },
        })
```

Additionally, **logs** and **events** are then iterated in order to parse and verify them.

```
events := make([]*types.EVMEvent, 0, len(logs))
    for _, l := range logs {
        topics := make([][]byte, 0, len(l.Topics))
        for _, t := range l.Topics {
            topics = append(topics, t.Bytes())
        }
        events = append(events, &types.EVMEvent{
            Address: l.Address.Bytes(),
            Topics:  topics,
            Data:    l.Data,
        })
    }

    for _, event := range events {
        if err := event.Verify(); err != nil {
            return nil, errors.Wrap(err, "verify event")
        }
    }
```

A malicious actor could exploit this mechanism by deploying a contract that emits a large number of events. This would significantly inflate the computational cost of filtering logs.

While this issue is generally inconsequential in off-chain operations, its occurrence within the *PrepareProposal()* ABCI++ handler presents a unique risk. The proposer operates within a constrained time frame, governed by the CometBFT consensus timeouts (see documentation).

Excessive processing time caused by this attack could lead to the proposer missing their round and incurring penalties. In extreme scenarios, the attack could slow down the chain or even cause it to halt.

Recommended mitigation

It is recommended optimizing the log retrieval process to mitigate the computational overhead. This may involve implementing pre-filtering mechanisms in Geth to reduce the complexity of *FilterLogs()*.

Team response

The client acknowledges the issue and states that the process of retrieving event logs with filters from a Geth client is based on a bloom filter.

Consequently, even in scenarios where a block contains numerous malicious events, the computational overhead for retrieving logs is relatively minimal.

TRST-L-4 User errors in staking or validator creation leads to loss of funds

- **Category:** Validation flaws
- **Source:** client/x/evmstaking/keeper/keeper.go
- **Status:** Acknowledged

Description

In the *stake()* and *CreateValidator()* functions of the **IPTokenStaking** contract, tokens are transferred from the **sender**, burned, and, respectively, a **Deposit** or **CreateValidator** event is emitted.

```
    function _stake(
        bytes calldata delegatorUncmpPubkey,
        bytes calldata validatorUncmpPubkey,
        IIPTokenStaking.StakingPeriod stakingPeriod,
        bytes calldata data
    ) internal returns (uint256) {
        // This can't be tested from Foundry (Solidity), but can be
triggered from js/rpc
        require(stakingPeriod <= IIPTokenStaking.StakingPeriod.LONG,
"IPTokenStaking: Invalid staking period");
        (uint256 stakeAmount, uint256 remainder) =
roundedStakeAmount(msg.value);
        require(stakeAmount >= minStakeAmount, "IPTokenStaking: Stake amount
under min");

        uint256 delegationId = 0;
        if (stakingPeriod != IIPTokenStaking.StakingPeriod.FLEXIBLE) {
            delegationId = ++_delegationIdCounter;
        }
        emit Deposit(
            delegatorUncmpPubkey,
            validatorUncmpPubkey,
```

```
            stakeAmount,
            uint8(stakingPeriod),
            delegationId,
            msg.sender,
            data
        );
        // We burn staked tokens
        payable(address(0)).transfer(stakeAmount);
```

```
    function _createValidator(
        bytes calldata validatorUncmpPubkey,
        string memory moniker,
        uint32 commissionRate,
        uint32 maxCommissionRate,
        uint32 maxCommissionChangeRate,
        bool supportsUnlocked,
        bytes calldata data
    ) internal {
        (uint256 stakeAmount, uint256 remainder) =
roundedStakeAmount(msg.value);
        require(stakeAmount >= minStakeAmount, "IPTokenStaking: Stake amount
under min");
        require(commissionRate >= minCommissionRate, "IPTokenStaking:
Commission rate under min");
        require(commissionRate <= maxCommissionRate, "IPTokenStaking:
Commission rate over max");
        payable(address(0)).transfer(stakeAmount);
        emit CreateValidator(
```

This event is subsequently processed on the Cosmos chain as part of the Geth payload during *PrepareProposal()*.

However, the *ExecutePayload()* message handler processes these events individually without validating that all events are successfully executed. This lack of robust event validation creates a risk where staking operations can fail on the Cosmos chain, while the sender's tokens remain irreversibly burned in the Solidity contract.

For example, if a user mistakenly stakes tokens to a nonexistent validator, the staking operation fails on the Cosmos chain, yet the tokens are still burned. Similarly, when creating a validator, submitting incorrect data (such as sending a duplicate transaction) could lead to the loss of all funds.

Without a rollback mechanism in the Solidity contract to restore burned tokens, users are left without recourse, and their funds are effectively lost.

Recommended mitigation

It is recommended to implement a rollback mechanism on the Solidity side to restore burned tokens in cases where the staking operation fails on the Cosmos chain would protect users from losing assets due to processing errors.

Team response

The client acknowledges the issue and states that they do not plan to support refunding tokens at the L1 level, as this is akin to users transferring tokens to an incorrect address. The client emphasizes that it is not their responsibility to recover such funds.

However, they commit to providing CLI and UI tools to help users reduce the likelihood of making such mistakes.

TRST-L-5 Misalignment between minimum required amount parameters on Solidity and Cosmos codebases

- **Category:** Logical flaws
- **Source:** client/x/evmstaking/keeper/validator.go
- **Status:** Acknowledged

Description

In the **IPTokenStaking** contract, the *setMinStakeAmount()* function allows the contract **owner** to dynamically adjust the minimum stake required for validator creation, staking, or redelegation. This flexibility enables the staking requirements to be updated as needed.

```
    function setMinStakeAmount(uint256 newMinStakeAmount) external onlyOwner
{
        _setMinStakeAmount(newMinStakeAmount);
    }
```

Similarly the *setMinCommissionRate()* function allows the contract **owner** to dynamically adjust the minimum commission rate for validators.

```
    function setMinCommissionRate(uint256 newValue) external onlyOwner {
        _setMinCommissionRate(newValue);
    }
```

However, the Cosmos chain enforces in staking operations its own **minDelegation** and **minCommissionRate** parameters.

```
    minSelfDelegation, err := k.stakingKeeper.MinDelegation(ctx)
    if err != nil {
        return errors.Wrap(err, "get min self delegation")
    }

    // Validator does not exist, create validator with self-delegation.
    msg, err := stypes.NewMsgCreateValidator(
        validatorAddr.String(),
        validatorPubkey,
        amountCoin,
        stypes.Description{Moniker: moniker},
        stypes.NewCommissionRates(
```

```
            // Divide these decimals by 100 to convert from basis points to
decimal. Will cut off decimal as the rates are integers.

math.LegacyNewDec(int64(ev.CommissionRate)).Quo(math.LegacyNewDec(10000)),

math.LegacyNewDec(int64(ev.MaxCommissionRate)).Quo(math.LegacyNewDec(10000))
,

math.LegacyNewDec(int64(ev.MaxCommissionChangeRate)).Quo(math.LegacyNewDec(1
0000)),
        ),
        minSelfDelegation, // make minimum self delegation align with
minimum delegation amount
        int32(ev.SupportsUnlocked),
    )
```

Unlike the dynamically adjustable parameter in the smart contract, the Cosmos **minDelegation** is static and cannot be updated dynamically due to the chain's reliance only on **ExecutePayload** messages.

This misalignment creates a risk of operational issues. If the **minStakeAmount** or **minCommissionRate** on the Solidity contract is adjusted without corresponding changes on the Cosmos side, staking operations, including validator creation and redelegation, may fail due to inconsistent requirements between the two environments.

Recommended mitigation

It is recommended to remove the minimum delegation check from the Cosmos chain to align staking requirements across both environments.

Team response

The client acknowledges awareness of the potential issue and states that this is a deliberate trade-off made to enhance the user experience.

TRST-L-6 Lack of CancelUpgrade message support

- **Category:** Functionality issues
- **Source:** client/x/evmengine/keeper/upgrades.go
- **Status:** Fixed

Description

The *ProcessUpgradeEvents()* function within the evmengine module identifies **SoftwareUpgrade** events and initializes upgrade plans in the Cosmos SDK's upgrade module.

```
        switch ethlog.Topics[0] {
        case types.SoftwareUpgradeEvent.ID:
            ev, err := k.upgradeContract.ParseSoftwareUpgrade(ethlog)
            if err != nil {
```

```
                clog.Error(ctx, "Failed to parse SubmitProposal log", err)
                continue
            }
            if err = k.ProcessSoftwareUpgrade(ctx, ev); err != nil {
                clog.Error(ctx, "Failed to process submit proposal", err)
                continue
            }
        }
    }
```

However, unlike the standard functionality of the Cosmos SDK, this implementation does not support **CancelUpgrade** messages. This limitation prevents the revocation of an upgrade plan if circumstances require cancellation, potentially exposing the system to unwanted upgrades or operational inconsistencies.

Recommended mitigation

It is recommended to modify the *ProcessUpgradeEvents()* function to handle **CancelUpgrade** messages. This would provide comprehensive upgrade management and mitigate risks associated with undesirable or erroneous upgrade plans.

Team response

Fixed.

Mitigation review

Fixed. The client has implemented a solution to address the issue by adding a handler for the **CancelUpgrade** message in the evmengine module. This enhancement ensures that the Cosmos layer can now effectively receive and process **CancelUpgrade** messages emitted by the Solidity contract.

TRST-L-7 Missing validation for token type during delegation

- **Category:** Validation issues
- **Source:** client/x/evmstaking/keeper/deposit.go
- **Status:** Acknowledged

Description

When creating a validator using the *CreateValidator()* function, the creator can pass the **supportsUnlocked** parameter to specify whether the validator supports unlocked tokens.

```
    function createValidator(
        bytes calldata validatorUncmpPubkey,
        string calldata moniker,
        uint32 commissionRate,
        uint32 maxCommissionRate,
        uint32 maxCommissionChangeRate,
        bool supportsUnlocked,
        bytes calldata data
```

```
    ) external payable
verifyUncmpPubkeyWithExpectedAddress(validatorUncmpPubkey, msg.sender)
nonReentrant {
        _createValidator(
            validatorUncmpPubkey,
            moniker,
            commissionRate,
            maxCommissionRate,
            maxCommissionChangeRate,
            supportsUnlocked,
            data
        );
    }
```

However, during token delegation, there is no mechanism to ensure that the delegated tokens align with the validator's specified token type.

This lack of validation could result in delegations involving unsupported token types, potentially causing inconsistencies or unexpected behavior.

Recommended mitigation

It is recommended to implement a validation check during the delegation process to ensure that the tokens provided match the validator's specified token type.

Team response

The client acknowledges the issue and states that the check will not be incorporated into the contract because the contract does not serve as the source of truth for validator states. Instead, the enforcement of the token check will be handled through legal contracts.

Additional recommendations

TRST-R-1 Handling of empty log topics

In tx.go, it provides a function *ToEthLog()* to convert an **EVMEvent** to an Ethereum log. If the **Topics** array is empty, an error is returned:

```
func (l *EVMEvent) ToEthLog() (ethtypes.Log, error) {
  if l == nil {
    return ethtypes.Log{}, errors.New("nil log")
  } else if len(l.Topics) == 0 {
    return ethtypes.Log{}, errors.New("empty topics")
  }
```

Events with no topics (anonymous events) are valid logs that can be emitted with the LOG0 opcode. While these are not used in Story's smart contracts and the parsing logic is therefore sufficient, not returning an error (but an empty array) for such events can avoid problems in future updates where this library is used for parsing arbitrary, valid logs.

Team response

Fixed.

Mitigation review

Fixed. The client addressed the issue by adding detailed comments to the codebase to clarify the behavior of the function.

TRST-R-2 Usage of Cosmos ORM singleton

The db.go code includes a custom implementation which ensures that **ExecutionHead** remains a singleton, i.e. contains one element at most. Cosmos ORM natively provides a singleton option (https://docs.cosmos.network/main/build/packages/orm#singletons) that could be used in evmengine.proto. Consider using this option, which would allow the removal of the custom implementation.

Team response

The client acknowledges the issue and states that, while they appreciate the suggestion, they will not adopt it at this time. However, they may consider addressing it in the future.

TRST-R-3 Wrong error message

The function *ExecutionHead.getExecutionHead()* returns the error message "update execution head", although it should be "get execution head". Consider changing the message.

Team response

Fixed.

Mitigation review

Fixed. The client resolved the issue by modifying the error messages as suggested, improving their clarity and ensuring better communication of issues.

TRST-R-4 Open TODOs

The codebase contains many open TODO items. Consider addressing these before the release and/or moving items that are only planned long-term to a dedicated task tracking solution.

Team response

Fixed.

Mitigation review

Fixed. The client resolved the issue by resolving pending TODOs.

TRST-R-5 Unimplemented inflation parameter update

The whitepaper mentions:

"For subsequent years, the number of emitted tokens will be controlled by an emissions algorithm whose parameters may be updated via governance or subject to change via hard forks."

Currently, it would only be possible to change these parameters via a hard fork, as there is no update handler that could be integrated with a governance contract / system. If this is still intended, consider implementing such an update procedure (for instance via EVM events that are emitted from a dedicated update contract and processed in the L1).

Team response

The client acknowledges the issue and states that they intentionally decided not to use an EVM event and instead opted for a hard fork to reduce centralization. The client further notes that, in the future, they plan to take a similar approach to remove the current upgrade event as well.

Centralization risks

TRST-CR-1 Singularity Period

During the singularity period, only 8 trusted validators will be in the active validator set. Slashing or jailing is explicitly disabled. Therefore, the usual incentives for acting honestly are not in place and a validator could for instance double sign without any repercussions.

Systemic risks

TRST-SR-1 Dependency on third-party software

Similarly to other blockchains, Story uses third-party software such as CometBFT, geth, or gogoproto. While the dependencies are carefully chosen and only battle-tested software is used, there is always the risk of an issue in one of these dependencies (such as the CometBFT issue that is documented in this report). Depending on the details, such an issue could have a significant impact on the chain and its applications.

TRST-SR-2 CometBFT safety assumptions

Because Story is using CometBFT, all of its guarantees and limitations regarding safety, liveness, fork attacks, or censorship attacks also apply to Story: https://docs.cometbft.com/main/spec/consensus/consensus