Making Web3 Space Safer for Everyone

KALOS

RIGO

Security Assessment

Published on : 2 June. 2023
Version v1.1

Security Report Published by KALOS

v1.1 2 June. 2023

Auditor : Jade, Jinu

hojung han

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	5	5	-	-
High	7	7	-	-
Medium	-	-	-	-
Low	2	2	-	-
Tips	5	5	-	-

TABLE OF CONTENTS

ABOUT US

Making Web3 Space Safer for Everyone

KALOS is a flagship service of HAECHI LABS, the leader of the global blockchain industry. We bring together the best Web2 and Web3 experts. Security Researchers with expertise in cryptography, leaders of the global best hacker team, and blockchain/smart contract experts are responsible for securing your Web3 service.

Having secured $60B crypto assets on over 400 main-nets, Defi protocols, NFT services, P2E, and Bridges, KALOS is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

Inquiries: audit@kalos.xyz
Website: https://kalos.xyz

Executive Summary

Purpose of this report

This report was prepared to audit the security of the project developed by the RIGO team. KALOS conducted the audit focusing on whether the system created by the RIGO team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the project.

In detail, we have focused on the following

- Denial of Service
- Freezing of User Assets
- Theft of User Assets
- Double Spend Bug
- Tamper/manipulate blockchain history to invalidate transactions
- Undermine consensus mechanism to split the chain
- Unhandled Exceptions

Codebase Submitted for the Audit

The codes used in this Audit can be found on GitHub (https://github.com/rigochain/rigo-go).

The commit of the code used for this Audit is “31e98792ea106f9cfa54952a59ec328b3bd40936”,

Audit Timeline

Date	Event
2023/05/11	Audit Initiation (RIGO)
2023/05/24	Delivery of v1.0 report.
2023/06/02	Delivery of v1.1 report.

Findings

KALOS found 5 Critical, 7 High and 2 Low severity issues.

Severity	Issue	Status
High	Proposal Voting failed due to incorrect block number and timestamp in CheckTx function	(Fixed - v1.1)
High	Validators subject to slashing can temporarily use Voting Power calculated with their balance before slashing	(Fixed - v1.1)
Critical	Dos vulnerability that prevents block creation	(Fixed - v1.1)
Critical	A malicious user could unstake another validator despite not being authorized	(Fixed - v1.1)
High	Inadequate lazyRewardBlocks Value in Current Initial GovRule Results in Insufficient Unstake Waiting Time	(Fixed - v1.1)
Critical	Native Token balance manipulation due to wrong StateDBWrapper implementation	(Fixed - v1.1)
Critical	Failure to Restore Native Token Balance upon Transferring and Executing Transaction Revert	(Fixed - v1.1)
Tips	Incorrect Txhash Setting in Event Log	(Fixed - v1.1)
High	Validator Slashing Does Not Impact Delegated User Assets	(Fixed - v1.1)
High	Block Timestamp is incorrectly set to nanoseconds	(Fixed - v1.1)
Tips	Overflow Risk in EndVotingHeight and ApplyingHeight within Governance Module Proposals	(Fixed - v1.1)
High	Incomplete Update of Staking Object with GovRule Changes through Voting	(Fixed - v1.1)
Tips	Checking TRX_STAKING type transaction in wrong place	(Fixed - v1.1)

Tips	No Block Reward for Empty Blocks	(Fixed - v1.1)
Critical	DoS via malicious GovRule data manipulation	(Fixed - v1.1)
Low	Insufficient Iterations in PBKDF2 for Private Keys	(Fixed - v1.1)
Low	Data Inconsistency Caused by Deleted and Re-created Ledger Entries	(Fixed - v1.1)
High	Insufficient Minimum Staking Requirement for Validators	(Fixed - v1.1)
Tips	Insufficient Token Distribution Explanation and Misalignment with Block Rewards-Based Distribution Design	(Fixed - v1.1)

Remarks

If malicious entities own more than 2/3 of the total voting power within a governance, a malicious proposal can be passed, causing damage to the RIGO Ecosystem.

OVERVIEW

Protocol overview

• **Account**

The Account module seems to be responsible for managing accounts and their related operations within the system. It includes functionalities for querying account information, transferring funds between accounts, and committing changes to the underlying account data.

• **Stake**

The Stake Module offers users the functionality of Staking and Unstaking. Users can participate in the consensus process of the Chain and receive Staking rewards through these features. This module enables users to actively engage in the protocol's consensus mechanism while earning Staking incentives.

• **Gov**

The Governance Module offers Validators the ability to propose changes to the Governance Rules through Proposals. It provides them with the functionality to suggest and implement modifications to the protocol's governance mechanisms. On the other hand, regular users have the capability to delegate their Voting Power and participate in voting. This module empowers users to delegate their influence and actively engage in decision-making processes within the protocol by utilizing their Voting Power.

• **VM**

The VM Module provides EVM compatibility within the RIGO ecosystem, enabling users to execute Smart Contracts that were originally designed to operate on EVM-compatible chains. By offering this functionality, the module allows seamless integration of existing Smart Contracts, ensuring their compatibility and functionality within the RIGO ecosystem. Users can leverage the VM Module to run and interact with EVM-compatible Smart Contracts, expanding the capabilities and possibilities of decentralized applications within the protocol.

Scope

```
├── cmd
│   ├── commands
│   │   ├── init.go
│   │   ├── reset.go
│   │   ├── root.go
│   │   ├── root_test.go
│   │   ├── run_node.go
│   │   ├── show_node_id.go
│   │   ├── show_wallet_key.go
│   │   └── version.go
│   ├── config
│   │   └── config.go
│   ├── main.go
│   └── version
│       └── version.go
├── ctrlers
│   ├── account
│   │   ├── account_test.go
│   │   ├── acct_codec.go
│   │   ├── ctrler.go
│   │   ├── ctrler_test.go
│   │   └── query.go
│   ├── gov
│   │   ├── 0_ctrler_test.go
│   │   ├── 1_proposal_test.go
│   │   ├── 2_voting_test.go
│   │   ├── 9_punish_test.go
│   │   ├── ctrler.go
│   │   ├── helper_test.go
│   │   ├── proposal
│   │   │   ├── header.go
│   │   │   ├── option.go
│   │   │   └── proposal.go
│   │   └── query.go
│   ├── stake
│   │   ├── ctrler.go
│   │   ├── ctrler_test.go
│   │   ├── delegatee.go
│   │   ├── delegatee_test.go
│   │   ├── helpers_test.go
│   │   ├── query.go
│   │   ├── stake.go
```

```
│   │   ├── stake_test.go
│   │   └── validators_test.go
│   ├── types
│   │   ├── account.go
│   │   ├── account.pb.go
│   │   ├── block_ctx.go
│   │   ├── gov_rule.go
│   │   ├── gov_rule.pb.go
│   │   ├── gov_rule_test.go
│   │   ├── handlers.go
│   │   ├── trx.go
│   │   ├── trx.pb.go
│   │   ├── trx_contract.go
│   │   ├── trx_ctx.go
│   │   ├── trx_proposal.go
│   │   ├── trx_staking.go
│   │   ├── trx_test.go
│   │   ├── trx_transfer.go
│   │   ├── trx_voting.go
│   │   └── vm.go
│   └── vm
│       └── evm
│           ├── block_ctx.go
│           ├── ctrler.go
│           ├── erc20_test.go
│           ├── erc20_test_contract.json
│           ├── query.go
│           └── statedb.go
├── genesis
│   ├── genesis.go
│   ├── genesis_devnet.go
│   ├── genesis_mainnet.go
│   └── genesis_testnet.go
├── ledger
│   ├── finality_ledger.go
│   ├── finality_ledger_test.go
│   ├── helpers.go
│   ├── mem_items.go
│   ├── mem_ledger.go
│   ├── simple_ledger.go
│   ├── simple_ledger_test.go
│   ├── sliceMap.go
│   └── types.go
├── libs
```

```
│   ├── file_io.go
│   ├── path.go
│   ├── sprompt.go
│   ├── utils.go
│   └── web3
│       ├── http_provider.go
│       ├── rigo_web3.go
│       ├── rpc.go
│       ├── subscriber.go
│       ├── trx.go
│       ├── types
│       │   ├── jsonrpc.go
│       │   ├── provider.go
│       │   └── tx_result.go
│       ├── vm
│       │   └── evm_contract.go
│       └── wallet.go
├── node
│   ├── app.go
│   ├── client.go
│   ├── meta_db.go
│   ├── node.go
│   ├── query.go
│   └── trx_executor.go
├── protos
│   ├── account.proto
│   ├── gov_rule.proto
│   └── trx.proto
├── rpc
│   ├── functions.go
│   ├── responses.go
│   └── routes.go
└── types
    ├── address.go
    ├── asset.go
    ├── asset_test.go
    ├── bytes
    │   ├── hex_bytes.go
    │   └── rand_bytes.go
    ├── crypto
    │   ├── crypto.go
    │   ├── crypto_test.go
    │   ├── sfile_pv.go
    │   ├── wallet_key.go
```

```
│   └── wallet_key_test.go
├── encodable.go
└── xerrors
    └── xerror.go
```

Access Role

Access roles play a crucial role in the functionality and governance of the RIGO.
This report focuses on two distinct access roles within the RIGO
The key differentiation between these roles lies in their ability to submit proposals to the Governance Module.

Validators - Validators hold a privileged access role within the RIGO ecosystem. Their primary responsibility is to validate and verify transactions, ensuring the integrity and security of the network. In addition to their transaction validation role, Validators have the authority to submit proposals to the Governance Module.

Normal Users - Normal Users represent the majority of participants within the RIGO ecosystem. They have the ability to interact with the RIGO, utilize its features, and conduct transactions. However, Normal Users do not possess the privilege to submit proposals to the Governance Module.

FINDINGS

1. Proposal Voting failed due to an incorrect block number and timestamp in the CheckTx function

ID: RIGO-01
Type: Logic Error
File: node/app.go

Severity: High
Difficulty: low

Issue

When validating a transaction in CheckTx, the block number and block timestamp are not specified correctly, causing the transaction to fail.

```
func (ctrler *RigoApp) CheckTx(req abcitypes.RequestCheckTx) abcitypes.ResponseCheckTx {
	ctrler.mtx.Lock()
	defer ctrler.mtx.Unlock()

	switch req.Type {
	case abcitypes.CheckTxType_New:
		txctx, xerr := types2.NewTrxContext(req.Tx,
			0, //ctrler.currBlockCtx.Height()+int64(1), <-- here
			0, <-- here
			false,
...
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/node/app.go#L206-L216

The code below will cause the transaction to fail.

```
func (ctrler *GovCtrler) ValidateTrx(ctx *ctrlertypes.TrxContext) xerrors.XError {
	...
	case ctrlertypes.TRX_VOTING:
		if bytes.Compare(ctx.Tx.To, types.ZeroAddress()) != 0 {
			return xerrors.ErrInvalidTrxPayloadParams.Wrap(errors.New("wrong
address: the 'to' field in TRX_VOTING should be zero address"))
		}
		// check tx type
		txpayload, ok := ctx.Tx.Payload.(*ctrlertypes.TrxPayloadVoting)
		if !ok {
			return xerrors.ErrInvalidTrxPayloadType
		}

		// check already exist
		prop, xerr := getProposal(txpayload.TxHash.Array32())
		if xerr != nil {
			return xerr
```

```
                }
                if prop.IsVoter(ctx.Tx.From) == false {
                        return xerrors.ErrNoRight
                }

                // check choice validation
                if txpayload.Choice < 0 || txpayload.Choice >= int32(len(prop.Options)) {
                        return xerrors.ErrInvalidTrxPayloadParams
                }

                // check end height
                if ctx.Height > prop.EndVotingHeight ||
                        ctx.Height < prop.StartVotingHeight {
                        return xerrors.ErrNotVotingPeriod
                }
        default:
                return xerrors.ErrUnknownTrxType
        }

        return nil
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/gov/ctrler.go#L147-L181

The Block Height is less than the StartVotingHeight because the Block Height is always set to 0.

So the vote transaction will always fail.

Recommendation

We recommend using the appropriate block timestamp and block number for CheckTx.

Fix Comment

As we recommended, the project team modified the code.

2. Validators subject to slashing can temporarily use Voting Power calculated with their balance before slashing

ID: RIGO-02
Type: Logic Error
File: ctrlers/gov/ctrler.go

Severity: High
Difficulty: low

Issue

There is an issue where validators who have been slashed for cheating temporarily have the voting power they had before being slashed.

The following function is executed when a proposal is created.

```
func (ctrler *GovCtrler) execProposing(ctx *ctrlertypes.TrxContext) xerrors.XError {

	setProposal := ctrler.proposalLedger.Set
	if ctx.Exec {
		setProposal = ctrler.proposalLedger.SetFinality
	}

	txpayload, _ := ctx.Tx.Payload.(*ctrlertypes.TrxPayloadProposal)

	voters := make(map[string]*proposal.Voter)
	vals, totalVotingPower := ctx.StakeHandler.Validators()
	for _, v := range vals { // <-- point!!
		voters[types.Address(v.Address).String()] = &proposal.Voter{
			Addr:   v.Address,
			Power:  v.Power,
			Choice: -1, // not choice
		}
	}

	prop := proposal.NewGovProposal(ctx.TxHash, txpayload.OptType,
		txpayload.StartVotingHeight, txpayload.VotingPeriodBlocks,
ctrler.LazyApplyingBlocks(),
		totalVotingPower, voters, txpayload.Options...)

	if xerr := setProposal(prop); xerr != nil {
		return xerr
	}

	return nil
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/gov/ctrler.go#L197-L225

In the above code, the voter is determined when the proposal is created, and the voting power is also determined, so it is treated as a kind of snapshot.
In contrast, the code for slashing only subtracts the voting power of each validator, not the voting power that is already stored in the proposed proposal as a snapshot.

```
// BeginBlock are called in RigoApp::BeginBlock
func (ctrler *StakeCtrler) BeginBlock(blockCtx *ctrlertypes.BlockContext)
([]abcitypes.Event, xerrors.XError) {
	var evts []abcitypes.Event

	byzantines := blockCtx.BlockInfo().ByzantineValidators
	if byzantines != nil {
		ctrler.logger.Debug("Byzantine validators is found", "count",
len(byzantines))
		for _, evi := range byzantines {
			if slashed, xerr := ctrler.doPunish(
				&evi, blockCtx.GovHandler.SlashRatio(),
				blockCtx.GovHandler.AmountPerPower(),
				blockCtx.GovHandler.RewardPerPower()); xerr != nil {
				ctrler.logger.Error("Error when punishing",
					"byzantine", types.Address(evi.Validator.Address),
					"evidenceType",
abcitypes.EvidenceType_name[int32(evi.Type)])
			} else {
				...
			}
		}
	}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/stake/ctrler.go#L84-L118

Recommendation

We recommend subtracting voting power from snapshots of proposals that have already been proposed.

Fix Comment

The snapshot has been slashed, but not the vote value, which is already voted.

The vote value was not slashed, but the MajorityPower and TotalVotingPower were, allowing the proposal to pass with less voting power.

This is now patched.

3. Dos vulnerability that prevents block creation

ID: RIGO-03
Type: Logic Error
File: node/app.go

Severity: Critical
Difficulty: low

Issue

An excessive GasLimit per block is set.

And the EVM does not deduct the gas from the sender's balance as long as the opcode is executed, and the sender can pay as much gas as they want.

Because of this, we confirm that there is a DOS that can prevent block creation at a small cost if it keeps shooting transactions that are in an excessive loop.

Below is the vulnerable code.

```
func evmBlockContext(sender common.Address, bn int64, tm int64) vm.BlockContext {
    return vm.BlockContext{
        CanTransfer: CanTransfer,
        Transfer:    Transfer,
        GetHash:     GetHash,
        Coinbase:    sender,
        BlockNumber: big.NewInt(bn),
        Time:        big.NewInt(tm),
        Difficulty:  big.NewInt(1),
        BaseFee:     big.NewInt(0),
        GasLimit:    gasLimit * 10_000, //10_000_000_000_000_000_000,
    }
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/vm/evm/block_ctx.go#L32-L44

```
func (ctrler *EVMCtrler) ExecuteTrx(ctx *ctrlertypes.TrxContext) xerrors.XError {
    ...
    ret, xerr := ctrler.execVM(
        ctx.Tx.From,
        [ctx.Tx.To](http://ctx.tx.to/),
        ctx.Tx.Nonce,
        ctx.Tx.Amount,
        ctx.Tx.Payload.(*ctrlertypes.TrxPayloadContract).Data,
        ctx.Height,
        ctx.BlockTime,
    )
    ...
    ctx.RetData = ret.ReturnData
```

```
    // vulnerable code
    ctx.GasUsed = ctx.Tx.Gas //new([uint256.Int](http://uint256.int/)).Add(ctx.GasUsed,
uint256.NewInt(ret.UsedGas))
    ctrler.stateDBWrapper.SubBalance(ctx.Tx.From.Array20(), ctx.GasUsed.ToBig())

    return nil
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/vm/evm/ctrler.go#L133-L165

Recommendation

We recommend setting `ctx.GasUsed` to `new(uint256.Int).Add(ctx.GasUsed, uint256.NewInt(ret.UsedGas))`, not `ctx.Tx.Gas` and setting `GasLimit` to 25000000, which is smaller than that. (25000000 is referenced to the value of Chronos Chain based on cosmos-sdk/tendermint)

Additionally, there is no code to deduct the gas used by the EVM from the Transaction Sender's balance. This needs to be added.

Fix Comment

As we recommended, the project team modified the code. Also fixed gaspool to be applied per block rather than per transaction.

4. A malicious user could unstake another validator despite not being authorized

ID: RIGO-04
Type: Logic Error
File: ctrlers/stake/ctrler.go

Severity: Critical
Difficulty: low

Issue

When unstaking, there is no logic to check if the transaction sender is the same as the initially staked party.

This allows all validators to be unstaked by others at no cost.

Below is the code for unstake.

```
func (ctrler *StakeCtrler) execUnstaking(ctx *ctrlertypes.TrxContext) xerrors.XError {
	getDelegatee := ctrler.delegateeLedger.Get
	setUpdateDelegatee := ctrler.delegateeLedger.Set
	delDelegatee := ctrler.delegateeLedger.Del
	setUpdateFrozen := ctrler.frozenLedger.Set
	if ctx.Exec {
		getDelegatee = ctrler.delegateeLedger.GetFinality
		setUpdateDelegatee = ctrler.delegateeLedger.SetFinality
		delDelegatee = ctrler.delegateeLedger.DelFinality
		setUpdateFrozen = ctrler.frozenLedger.SetFinality
	}

	// find delegatee
	delegatee, xerr := getDelegatee(ledger.ToLedgerKey(ctx.Tx.To))
	if xerr != nil {
		return xerr
	}

	// delete the stake from a delegatee
	txhash := ctx.Tx.Payload.(*ctrlertypes.TrxPayloadUnstaking).TxHash
	if txhash == nil && len(txhash) != 32 {
		return xerrors.ErrInvalidTrxPayloadParams
	}
	s0 := delegatee.DelStake(txhash)
	if s0 == nil {
		return xerrors.ErrNotFoundStake
	}

	s0.RefundHeight = ctx.Height + ctx.GovHandler.LazyRewardBlocks()
	_ = setUpdateFrozen(s0) // add s0 to frozen ledger

	if delegatee.SelfPower == 0 {
		stakes := delegatee.DelAllStakes()
```

```
                for _, _s0 := range stakes {
                        _s0.RefundHeight = ctx.Height + ctx.GovHandler.LazyRewardBlocks()
                        _ = setUpdateFrozen(_s0) // add s0 to frozen ledger
                }
        }

        if delegatee.TotalPower == 0 {
                // this changed delegate will be committed at Commit()
                if _, xerr := delDelegatee(delegatee.Key()); xerr != nil {
                        return xerr
                }
        } else {
                // this changed delegate will be committed at Commit()
                if xerr := setUpdateDelegatee(delegatee); xerr != nil {
                        return xerr
                }
        }

        return nil
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/stake/ctrler.go#L207-L259

Recommendation

When unstaking, it is recommended to check if the transaction sender is the same party that originally staked.

Fix Comment

As we recommended, the project team modified the code.

5. Inadequate lazyRewardBlocks Value in Current Initial GovRule Results in Insufficient Unstake Waiting Time

ID: RIGO-05
Type: Logic Error
File: ctrlers/types/gov_rule.go

Severity: High
Difficulty: Medium

Issue

The code below shows the settings for the initial GovRule.

```
// ctrlers/types/gov_rules.go
func DefaultGovRule() *GovRule {
	return &GovRule{
		version:                0,
		maxValidatorCnt:        21,
		amountPerPower:         uint256.NewInt(1_000000000_000000000),
		rewardPerPower:         uint256.NewInt(1_000000000),
		lazyRewardBlocks:       20,
		lazyApplyingBlocks:     10,
		minTrxFee:              uint256.NewInt(10),
		minVotingPeriodBlocks:  259200,  // = 60 * 60 * 24 * 3, // 3 days
		maxVotingPeriodBlocks:  2592000, // = 60 * 60 * 24 * 30,    // 30 days
		minSelfStakeRatio:      50,      // 50%
		maxUpdatableStakeRatio: 30,      // 30%
		slashRatio:             50,      // 50%
	}
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/types/gov_rule.go#L34-L49

The code below shows that when an Unstake Transaction is executed, it delays the return of staking up to the height of the block at the time it was executed plus the GovRule's lazyRewardBlocks.

```
func (ctrler *StakeCtrler) EndBlock(ctx *ctrlertypes.BlockContext) ([]abcitypes.Event,
xerrors.XError) {
	ctrler.mtx.Lock()
	defer ctrler.mtx.Unlock()

	if ctx.TxsCnt() > 0 {
		if xerr := ctrler.doReward(ctx.BlockInfo().Header.Height,
ctx.BlockInfo().LastCommitInfo.Votes); xerr != nil {
			return nil, xerr
		}
	}
	if xerr := ctrler.unfreezingStakes(ctx.Height(), ctx.AcctHandler); xerr != nil {
```

```
                return nil, xerr
        }
        ctx.SetValUpdates(ctrler.updateValidators(int(ctx.GovHandler.MaxValidatorCnt())))

        return nil, nil
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/stake/ctrler.go#L261-L276

```
func (ctrler *StakeCtrler) unfreezingStakes(height int64, acctHandler
ctrlertypes.IAccountHandler) xerrors.XError {
        return ctrler.frozenLedger.IterateReadAllFinalityItems(func(s0 *Stake) xerrors.XError
{
                if s0.RefundHeight <= height {
                        // un-freezing s0
                        // return not only s0.ReceivedReward but also s0.Amount
                        xerr := acctHandler.Reward(
                                s0.From,
                                new(uint256.Int).Add(s0.Amount, s0.ReceivedReward),
                                true)
                        if xerr != nil {
                                return xerr
                        }

                        _, _ = ctrler.frozenLedger.DelFinality(ledger.ToLedgerKey(s0.TxHash))
                }
                return nil
        })
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/stake/ctrler.go#L310-L322

Due to the nature of Tendermint, if a malicious user cheats, the malicious user is not slashed immediately, but after some time, evidence is collected, and if it is recognized, the malicious user is slashed.

The number 20, set to lazyRewardBlocks in the initial govrule, is a tiny number considering that Tendermint generates a block every 2-3 seconds when 100 validators are connected.

Recommendation

We recommend setting lazyRewardBlocks to a higher number.

Fix Comment

As we recommended, the project team modified the code.
*(lazyRewardBlocks: 2592000, // = 60 * 60 * 24 * 30 => 30 days)*

6. Native Token balance manipulation due to wrong StateDBWrapper implementation

ID: RIGO-06
Type: Logic Error
File: ctrlers/vm/evm/statedb.go

Severity: Critical
Difficulty: Low

Issue

The code below is relevant to this issue.

```
func (s *StateDBWrapper) SubBalance(addr common.Address, amt *big.Int) {
	if acct := s.acctHandler.FindAccount(addr[:], true); acct != nil {
		if err := acct.SubBalance(uint256.MustFromBig(amt)); err != nil {
			panic(err)
		}
	}
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/vm/evm/statedb.go#L89-L97

```
func (s *StateDBWrapper) AddBalance(addr common.Address, amt *big.Int) {
	if acct := s.acctHandler.FindAccount(addr[:], true); acct != nil {
		if err := acct.AddBalance(uint256.MustFromBig(amt)); err != nil {
			panic(err)
		}
	}
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/vm/evm/statedb.go#L99-L107

```
func opSelfdestruct(pc *uint64, interpreter *EVMInterpreter, scope *ScopeContext) ([]byte,
error) {
	if interpreter.readOnly {
		return nil, ErrWriteProtection
	}
	beneficiary := scope.Stack.pop()
	balance := interpreter.evm.StateDB.GetBalance(scope.Contract.Address())
	interpreter.evm.StateDB.AddBalance(beneficiary.Bytes20(), balance)
	interpreter.evm.StateDB.Suicide(scope.Contract.Address())
	...
	return nil, errStopToken
}
```

https://github.com/ethereum/go-ethereum/blob/9231770811cda0473a7fa4e2bccc95bf62aae634/core/vm/instructions.go#L817-L830

```
func (s *StateDB) Suicide(addr common.Address) bool {
	stateObject := s.getStateObject(addr)
	if stateObject == nil {
		return false
	}
	s.journal.append(suicideChange{
		account:     &addr,
		prev:        stateObject.suicided,
		prevbalance: new(big.Int).Set(stateObject.Balance()),
	})
	stateObject.markSuicided()
	stateObject.data.Balance = new(big.Int) // <-- balance 초기화

	return true
}
```

https://github.com/ethereum/go-ethereum/blob/9231770811cda0473a7fa4e2bccc95bf62aae634/core/state/statedb.go#L461-L475

In EVM, `selfdestruct` is a built-in function that destroys the contract's code and sends all the Ethereum in the contract to the specified recipient.

The code that handles this in EVM is the `opSelfdestruct` and `Suicide` functions mentioned above.

The code is not deleted as soon as it is executed, but when the statedb is committed, so malicious user can execute `selfdestruct` multiple times in one transaction.

If you compile go-ethereum and run it on a node, it will initialize the balance in the suicide function even if the malicious user executes selfdestruct multiple times.

So, even if `opSelfdestruct` is executed multiple times and `AddBalance` is executed multiple times as a result, there is no problem because 0 ETH is added from the second `opSelfdestruct` call.

However, in rigo, balance is managed separately in the accounts module, and in the `Suicide` function, the balance is not initialized using the method provided by rigo's `StateDBWrapper`, but is initialized by directly accessing StateDB's `data.Balance`, so there is a problem that the balance of the account is not initialized properly, and if a malicious user uses `selfdestruct` multiple times, money can be copied.

Recommendation

We recommend using Ethermint based on the Cosmos SDK.

Fix Comment

We reviewed issues 6, 7, and 8 together because they were similar.
The project team modified the code differently than recommended. The project team modified the StateDBWrapper code to sync the state information of the EVM and RIGO. The project team modified the StateDBWrapper code to sync the state data of evm and rigo. Modifying the StateDBWrapper code also caused new security issues, but kalos and the project team communicated via Slack and GitHub to resolve the security issues.

7. Failure to Restore Native Token Balance upon Transferring and Executing Transaction Revert

ID: RIGO-07
Type: Logic Error
File: ctrlers/types/gov_rule.go

Severity: Critical
Difficulty: Low

Issue

The code for revert related to balance transfer is shown below.

```
func (ch balanceChange) revert(s *StateDB) {
	s.getStateObject(*ch.account).setBalance(ch.prev)
}
```

https://github.com/ethereum/go-ethereum/blob/9231770811cda0473a7fa4e2bccc95bf62aae634/core/state/journal.go#L189-L191

The reasons for this issue are similar to those mentioned in RIGO-6.

There is a problem that the balance of the account is not initialized properly because we are managing the Native Token Balance separately in the Account Module, and when we initialize the balance in the Suicide function, we access StateDB's data.Balance directly to initialize it instead of initializing it in the way provided by rigo's StateDBWrapper.

For the above reasons, a malicious user can freeze the Native Token Balance for all smart contracts existing in the rigo ecosystem.

Recommendation

We recommend using Ethermint based on the Cosmos SDK.

Fix Comment

The project team modified the code differently than recommended. The project team modified the StateDBWrapper code to sync the state information of the EVM and RIGO.

8. Incorrect Txhash Setting in Event Log

ID: RIGO-08
Type: Logic Error
File: ctrlers/vm/evm/statedb.go

Severity: Tips
Difficulty: -

Issue

The following code will match the logs and txhash of the occurrence.

```
// go-ethereum, core/state/statedb.go
func (s *StateDB) AddLog(log *types.Log) {
	s.journal.append(addLogChange{txhash: s.thash})

	log.TxHash = s.thash
	log.TxIndex = uint(s.txIndex)
	log.Index = s.logSize
	s.logs[s.thash] = append(s.logs[s.thash], log)
	s.logSize++
}
```

https://github.com/ethereum/go-ethereum/blob/9231770811cda0473a7fa4e2bccc95bf62aae634/core/state/statedb.go#L198-L206

Where s.thash is set in the code below can be seen.

```
func (s *StateDB) SetTxContext(thash common.Hash, ti int) {
	s.thash = thash
	s.txIndex = ti
}
```

https://github.com/ethereum/go-ethereum/blob/9231770811cda0473a7fa4e2bccc95bf62aae634/core/state/statedb.go#L947-L950

```
func (p *StateProcessor) Process(block *types.Block, statedb *state.StateDB, cfg vm.Config)
(types.Receipts, []*types.Log, uint64, error) {
	...
	for i, tx := range block.Transactions() {
		msg, err := tx.AsMessage(types.MakeSigner(p.config, header.Number),
header.BaseFee)
		...
		statedb.SetTxContext(tx.Hash(), i)
		receipt, err := applyTransaction(msg, p.config, gp, statedb, blockNumber,
blockHash, tx, usedGas, vmenv)
		...
	}
	...
}
```

https://github.com/ethereum/go-ethereum/blob/9231770811cda0473a7fa4e2bccc95bf62aae634/core/state_processor.go#L52-L101

In the above code, SetTxContext is called when a block is created in go-ethereum.

However, in rigo, the block creation is done by tendermint and in go-ethereum, SetTxContext is not executed because it only uses EVM.

This results in s.thash not being set properly.

Recommendation

We recommend using Ethermint based on the Cosmos SDK.

Fix Comment

The project team modified the code differently than recommended. The project team modified the code to call evm's StateDB.Prepare function and return event data in the form of abcitypes.Event so that events could be tracked per transaction.

9. Validator Slashing Does Not Impact Delegated User Assets

ID: RIGO-09
Type: Logic Error
File: ctrlers/stake/delegatee.go

Severity: High
Difficulty: Low

Issue

The provided code deducts assets from malicious users when a validator engages in negative behavior.

However, it fails to subtract the holdings of users delegated to the validator.

This loophole allows malicious validators to exploit the system by creating multiple wallets and evading slashing penalties.

Please note that the provided code snippet demonstrates the lack of deduction for delegated user assets when a validator is slashed.

```
func (delegatee *Delegatee) DoSlash(ratio int64, amtPerPower, rwdPerPower *uint256.Int)
int64 {
	delegatee.mtx.Lock()
	defer delegatee.mtx.Unlock()

	_p0 := uint256.NewInt(uint64(delegatee.SelfPower))
	_ = _p0.Mul(_p0, uint256.NewInt(uint64(ratio)))
	_ = _p0.Div(_p0, uint256.NewInt(uint64(100)))
	slashingPower := int64(_p0.Uint64())
	slashedPower := int64(0)

	var removingStakes []*Stake
	for _, s0 := range delegatee.Stakes {
		if s0.From.Compare(delegatee.Addr) == 0 && s0.IsSelfStake() {
			if s0.Power <= slashingPower {
				slashingPower -= s0.Power
				slashedPower += s0.Power

				// power, amount is processed at out of loop
				removingStakes = append(removingStakes, s0)
			} else {
				s0.Power -= slashingPower
				slashedPower += slashingPower

				blocks := uint64(0)
				if s0.ReceivedReward.Sign() > 0 {
					blocks = new(uint256.Int).Div(s0.ReceivedReward,
```

```
s0.BlockRewardUnit).Uint64()
                                }

                                s0.Amount =
new(uint256.Int).Mul(uint256.NewInt(uint64(s0.Power)), amtPerPower)
                                s0.BlockRewardUnit = new(uint256.Int).Mul(rwdPerPower,
uint256.NewInt(uint64(s0.Power)))
                                if blocks > 0 {
                                        s0.ReceivedReward =
new(uint256.Int).Mul(s0.BlockRewardUnit, uint256.NewInt(blocks))
                                }

                                slashingPower = 0
                        }
                        if slashingPower == 0 {
                                break
                        }
                }
        }

        if removingStakes != nil {
                for _, s1 := range removingStakes {
                        _ = delegatee.delStakeByHash(s1.TxHash)
                }
        }

        ...
        return slashedPower
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/stake/delegatee.go#L296-L351

Recommendation

Users who have delegated their assets to a validator must also bear the asset deduction caused by slashing along with the validator.

Fix Comment

As we recommended, the project team modified the code.

10. Block Timestamp is incorrectly set to nanoseconds

ID: RIGO-10
Type: Logic Error
File: ctrlers/types/block_ctx.go

Severity: High
Difficulty: Low

Issue

The block.timestamp is set in nanoseconds, not seconds.

This means that if the EVM uses block.timestamp to determine how much to pay out in certain situations, it will likely calculate an excessively large value.

Below is the relevant code.

```
func (bctx *BlockContext) TimeNano() int64 {
	bctx.mtx.RLock()
	defer bctx.mtx.RUnlock()

	return bctx.blockInfo.Header.GetTime().UnixNano()
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/types/block_ctx.go#L82-L87

```
func (ctrler *RigoApp) deliverTxSync(req abcitypes.RequestDeliverTx)
abcitypes.ResponseDeliverTx {

	txctx, xerr := types2.NewTrxContext(req.Tx,
		ctrler.currBlockCtx.Height(),
		ctrler.currBlockCtx.TimeNano(),
		true,
		func(_txctx *types2.TrxContext) xerrors.XError {
			_txctx.TxIdx = ctrler.currBlockCtx.TxsCnt()
			ctrler.currBlockCtx.AddTxsCnt(1)

			_txctx.TrxGovHandler = ctrler.govCtrler
			_txctx.TrxAcctHandler = ctrler.acctCtrler
			_txctx.TrxStakeHandler = ctrler.stakeCtrler
			_txctx.TrxEVMHandler = ctrler.vmCtrler
			_txctx.GovHandler = ctrler.govCtrler
			_txctx.StakeHandler = ctrler.stakeCtrler
			return nil
		})
	if xerr != nil {
		....
	}
	xerr = ctrler.txExecutor.ExecuteSync(txctx)
	if xerr != nil {
		...
```

```
        } else {
                ...
        }
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/node/app.go#L274-L327

Recommendation

We recommend using second instead of nanosecond as the timestamp for the block.

Fix Comment

As we recommended, the project team modified the code.

11. Overflow Risk in EndVotingHeight and ApplyingHeight within Governance Module Proposals

ID: RIGO-11
Type: Logic Error
File: ctrlers/gov/proposal/proposal.go

Severity: Tips
Difficulty: -

Issue

The code below creates a Proposal for a new GovRule inside the Governance Module.

```
func (ctrler *GovCtrler) execProposing(ctx *ctrlertypes.TrxContext) xerrors.XError {

	setProposal := ctrler.proposalLedger.Set
	if ctx.Exec {
		setProposal = ctrler.proposalLedger.SetFinality
	}

	txpayload, _ := ctx.Tx.Payload.(*ctrlertypes.TrxPayloadProposal)

	voters := make(map[string]*proposal.Voter)
	vals, totalVotingPower := ctx.StakeHandler.Validators()
	for _, v := range vals {
		voters[types.Address(v.Address).String()] = &proposal.Voter{
			Addr:   v.Address,
			Power:  v.Power,
			Choice: proposal.NOT_CHOICE, // -1
		}
	}

	prop := proposal.NewGovProposal(ctx.TxHash, txpayload.OptType,
		txpayload.StartVotingHeight, txpayload.VotingPeriodBlocks,
ctrler.LazyApplyingBlocks(),
		totalVotingPower, voters, txpayload.Options...)

	if xerr := setProposal(prop); xerr != nil {
		return xerr
	}

	return nil
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/gov/ctrler.go#L197-L225

The code below creates a Proposal for a new GovRule inside the Governance Module.

If you look at the values of EndVotingHeight, ApplyingHeight in the NewGovProposal function that creates the GovRule object, there is no Overflow Check.

If a user specifies a startHeight that causes an overflow, the Proposal may end without allowing enough voting time.

```
func NewGovProposal(txhash bytes.HexBytes, optType int32, startHeight, votingBlocks,
lazyApplyingBlocks, totalVotingPower int64, voters map[string]*Voter, options ...[]byte)
*GovProposal {
	return &GovProposal{
		GovProposalHeader: GovProposalHeader{
			TxHash:            txhash,
			StartVotingHeight: startHeight,
			EndVotingHeight:   startHeight + votingBlocks,
			ApplyingHeight:    startHeight + votingBlocks + lazyApplyingBlocks,
			TotalVotingPower:  totalVotingPower,
			MajorityPower:     (totalVotingPower * 2) / 3,
			Voters:            voters,
			OptType:           optType,
		},
		Options:     NewVoteOptions(options...),
		MajorOption: nil,
	}
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/gov/proposal/proposal.go#L21-L36

Recommendation

We recommend adding overflow check code for EndVotingHeight and ApplyingHeight.

Fix Comment

As we recommended, the project team modified the code.

12. Incomplete Update of Staking Object with GovRule Changes through Voting

ID: RIGO-12
Type: Logic Error
File: ctrlers/gov/ctrler.go

Severity: High
Difficulty: High

Issue

The below code successfully updates the new GovRule.

However, the values of existing stake objects are not updated.

```
func (ctrler *GovCtrler) applyProposals(height int64) xerrors.XError {
	xerr := ctrler.frozenLedger.IterateReadAllItems(func(prop *proposal.GovProposal)
xerrors.XError {
		if prop.ApplyingHeight <= height {
			...
			if prop.MajorOption != nil {
				switch prop.OptType {
				case proposal.PROPOSAL_GOVRULE:
					newGovRule := &ctrlertypes.GovRule{}
					if err := json.Unmarshal(prop.MajorOption.Option(),
newGovRule); err != nil {
						return xerrors.From(err)
					}
					if xerr := ctrler.ruleLedger.SetFinality(newGovRule);
xerr != nil {
						return xerr
					}
					ctrler.newGovRule = newGovRule
				default:
					key := prop.Key()
					ctrler.logger.Debug("Apply proposal", "key(txHash)",
abytes.HexBytes(key[:]), "type", prop.OptType)
				}
			} else {
				ctrler.logger.Error("Apply proposal", "error", "major option is
nil")
			}
		}
		return nil
	})

	return xerr
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/gov/ctrler.go#L298-L327

```
func (ctrler *GovCtrler) Commit() ([]byte, int64, xerrors.XError) {
        ...
        if ctrler.newGovRule != nil {
                ctrler.GovRule = *ctrler.newGovRule
                ctrler.newGovRule = nil
                ....
        }
         ...
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/gov/ctrler.go#L329-L357

Recommendation

When the GovRule is updated, it is recommended to update the power and BlockRewardUnit based on the amount of the existing Stake Object.

Fix Comment

As we recommended, the project team modified the code. The AmoutPerPower value was deleted from the GovRule to make it static, and the reward was changed to calculate with the stake data every block.

13. Checking TRX_STAKING type transaction in wrong place

ID: RIGO-13
Type: Logic Error
File: ctrlers/stake/ctrler.go

Severity: Tips
Difficulty: -

Issue

The given code validates a Transaction of type TRX_STAKING in the Governance Module, not the Stake Module.

Below is the relevant code.

```
// ctrlers/stake/ctrler.go
func (ctrler *StakeCtrler) ValidateTrx(ctx *ctrlertypes.TrxContext) xerrors.XError {
	switch ctx.Tx.GetType() {
	case ctrlertypes.TRX_STAKING:
	case ctrlertypes.TRX_UNSTAKING:
	default:
		return xerrors.ErrUnknownTrxType
	}

	return nil
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/stake/ctrler.go#L136-L145

```
func (ctrler *GovCtrler) ValidateTrx(ctx *ctrlertypes.TrxContext) xerrors.XError {
	ctrler.mtx.RLock()
	defer ctrler.mtx.RUnlock()

	getProposal := ctrler.proposalLedger.Get
	if ctx.Exec {
		getProposal = ctrler.proposalLedger.GetFinality
	}

	// common validation for all trxs
	// check min tx fee
	if ctrler.MinTrxFee().Cmp(ctx.Tx.Gas) > 0 {
		return xerrors.ErrInsufficientFee
	}

	// validation by tx type
	switch ctx.Tx.GetType() {
	case ctrlertypes.TRX_STAKING:
		q, r := new(uint256.Int).DivMod(ctx.Tx.Amount, ctrler.AmountPerPower(),
new(uint256.Int))
```

```
                // `ctx.Tx.Amount` MUST be greater than or equal to
`ctrler.govHelper.AmountPerPower()`
                //    ==> q.Sign() > 0
                if q.Sign() <= 0 {
                        return xerrors.ErrInvalidTrx.Wrap(fmt.Errorf("wrong amount: it should
be greater than %v", ctrler.AmountPerPower()))
                }
                // `ctx.Tx.Amount` MUST be multiple to `ctrler.govHelper.AmountPerPower()`
                //    ==> r.Sign() == 0
                if r.Sign() != 0 {
                        return xerrors.ErrInvalidTrx.Wrap(fmt.Errorf("wrong amount: it should
be multiple of %v", ctrler.AmountPerPower()))
                }
        case ctrlertypes.TRX_PROPOSAL:
                if bytes.Compare(ctx.Tx.To, types.ZeroAddress()) != 0 {
                        return xerrors.ErrInvalidTrx.Wrap(errors.New("wrong address: the 'to'
field in TRX_PROPOSAL should be zero address"))
                }

                // check right
                if ctx.StakeHandler.IsValidator(ctx.Tx.From) == false {
                        return xerrors.ErrNoRight
                }

                // check tx type
                txpayload, ok := ctx.Tx.Payload.(*ctrlertypes.TrxPayloadProposal)
                if !ok {
                        return xerrors.ErrInvalidTrxPayloadType
                }

                // check already exist
                if prop, xerr := getProposal(ctx.TxHash.Array32()); xerr != nil && xerr !=
xerrors.ErrNotFoundResult {
                        return xerr
                } else if prop != nil {
                        return xerrors.ErrDuplicatedKey
                }

                // check start height
                if txpayload.StartVotingHeight <= ctx.Height {
                        return xerrors.ErrInvalidTrxPayloadParams
                }
                // check voting period
                if txpayload.VotingPeriodBlocks > ctrler.MaxVotingPeriodBlocks() ||
                        txpayload.VotingPeriodBlocks < ctrler.MinVotingPeriodBlocks() {
                        return xerrors.ErrInvalidTrxPayloadParams
                }
        case ctrlertypes.TRX_VOTING:
                if bytes.Compare(ctx.Tx.To, types.ZeroAddress()) != 0 {
                        return xerrors.ErrInvalidTrxPayloadParams.Wrap(errors.New("wrong
address: the 'to' field in TRX_VOTING should be zero address"))
                }
                // check tx type
                txpayload, ok := ctx.Tx.Payload.(*ctrlertypes.TrxPayloadVoting)
                if !ok {
                        return xerrors.ErrInvalidTrxPayloadType
```

```
                }

                // check already exist
                prop, xerr := getProposal(txpayload.TxHash.Array32())
                if xerr != nil {
                        return xerr
                }
                if prop.IsVoter(ctx.Tx.From) == false {
                        return xerrors.ErrNoRight
                }

                // check choice validation
                if txpayload.Choice < 0 || txpayload.Choice >= int32(len(prop.Options)) {
                        return xerrors.ErrInvalidTrxPayloadParams
                }

                // check end height
                if ctx.Height > prop.EndVotingHeight ||
                        ctx.Height < prop.StartVotingHeight {
                        return xerrors.ErrNotVotingPeriod
                }
        default:
                return xerrors.ErrUnknownTrxType
        }

        return nil
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/gov/ctrler.go#L86-L181

Recommendation

We recommend modifying the project code so that transactions of type TRX_STAKING are validated by the Stake module.

Fix Comment

As we recommended, the project team modified the code.

14. No Block Reward for Empty Blocks

ID: RIGO-14
Type: Logic Error
File: ctrlers/stake/ctrler.go

Severity: Tips
Difficulty: -

Issue

The code below does not call the doReward function, which pays the reward if the number of transactions in the block is zero.

Even if there are no transactions in the block, the reward must be paid to the stakers.

```
func (ctrler *StakeCtrler) EndBlock(ctx *ctrlertypes.BlockContext) ([]abcitypes.Event,
xerrors.XError) {
	ctrler.mtx.Lock()
	defer ctrler.mtx.Unlock()

	if ctx.TxsCnt() > 0 {
		if xerr := ctrler.doReward(ctx.BlockInfo().Header.Height,
ctx.BlockInfo().LastCommitInfo.Votes); xerr != nil {
			return nil, xerr
		}
	}
	if xerr := ctrler.unfreezingStakes(ctx.Height(), ctx.AcctHandler); xerr != nil {
		return nil, xerr
	}
	ctx.SetValUpdates(ctrler.updateValidators(int(ctx.GovHandler.MaxValidatorCnt())))

	return nil, nil
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/stake/ctrler.go#L261-L276

Recommendation

We recommend removing the if statement that checks if the number of transactions in the block is zero.

Even if there are no transactions in the block, you still need to reward stakers.

Fix Comment

As we recommended, the project team modified the code.

15. DoS via malicious GovRule data manipulation

ID: RIGO-15
Type: Logic Error
File: ctrlers/types/block_ctx.go

Severity: Critical
Difficulty: High

Issue

The GovRule struct in the rigo, utilizes the uint256.MustFromHex function to parse data during the UnmarshalJSON process. The uint256.MustFromHex function does not return an error value when there is an issue with the data but instead triggers a panic. Consequently, if invalid hex-formatted values are provided for the uint256 data used in GovRule, it can lead to a DoS scenario in the rigo.

```
// rigo-go/ctrlers/gov/ctrler.go#L298
func (ctrler *GovCtrler) applyProposals(height int64) xerrors.XError {
	xerr := ctrler.frozenLedger.IterateReadAllItems(func(prop *proposal.GovProposal)
xerrors.XError {
		if prop.ApplyingHeight <= height {
			if _, xerr := ctrler.frozenLedger.DelFinality(prop.Key()); xerr != nil
{
				return xerr
			}
			if prop.MajorOption != nil {
				switch prop.OptType {
				case proposal.PROPOSAL_GOVRULE:
					newGovRule := &ctrlertypes.GovRule{}
					if err := json.Unmarshal(prop.MajorOption.Option(),
newGovRule); err != nil {
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/gov/ctrler.go#L298-L327

```
// rigo-go/ctrlers/types/gov_rule.go#L187
func (r *GovRule) UnmarshalJSON(bz []byte) error {
	...
	r.amountPerPower = uint256.MustFromHex(tm.AmountPerPower)
	r.rewardPerPower = uint256.MustFromHex(tm.RewardPerPower)
	r.lazyRewardBlocks = tm.LazyRewardBlocks
	r.lazyApplyingBlocks = tm.LazyApplyingBlocks
	r.minTrxFee = uint256.MustFromHex(tm.MinTrxFee)
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ctrlers/types/gov_rule.go#L187-L223

```
// github.com/holiman/uint256@v1.2.2/conversion.go#L130
// MustFromHex is a convenience-constructor to create an Int from
// a hexadecimal string.
// Returns a new Int and panics if any error occurred.
```

```
func MustFromHex(hex string) *Int {
	var z Int
	if err := z.fromHex(hex); err != nil {
		panic(err)
	}
	return &z
}
```

https://github.com/holiman/uint256/blob/e42b95a00943c293245c6fe6b6855c483778e21b/conversion.go#L183-L189

Recommendation

We recommend changing the data parsing function in the GovRule struct to gracefully handle errors without causing panic. Use the FromHex function instead of the MustFromHex function and ensure proper error handling when errors occur.

Fix Comment

As we recommended, the project team modified the code.

16. Insufficient Iterations in PBKDF2 for Private Keys

ID: RIGO-16
Type: Logic Error
File: types/crypto/wallet_key.go

Severity: Low
Difficulty: High

Issue

The current implementation of encrypting private keys using PBKDF2 with few iterations poses a security risk.

A lower number of iterations can weaken the overall security strength of the encryption process, potentially making the private keys more susceptible to brute-force attacks.

The encryption process for storing private keys involves using PBKDF2 with relatively few iterations. The chosen iteration count in the current implementation is significantly lower than the recommended security standards.

According to the OWASP guidelines for 2023 (https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#pbkdf2), a minimum of 600,000 iterations is recommended for the PBKDF2 algorithm. However, only 20,000 iterations are being used in Rigo, which falls considerably short of the recommended threshold.

The number of iterations directly impacts the computational cost and time required for deriving the encryption key from the password. The higher iteration counts significantly increase the time and resources needed to perform a brute-force attack, thereby enhancing the security of the encrypted private keys.

```
// rigo-go/types/crypto/wallet_key.go#L60
func NewWalletKey(keyBytes, pass []byte) *WalletKey {
	var _pubKey, _prvKey []byte
	var _cipherTextParams *cipherTextParams
	var _dkParams *dkParams

	if pass != nil {
		salt := make([]byte, DKLEN)
		rand.Read(salt)
		iter := 20000 + int(binary.BigEndian.Uint16(append([]byte{0x00},
salt[:1]...)))
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/types/crypto/wallet_key.go#L60-L127

Recommendation

We recommend increasing the number of iterations in the PBKDF2 process to meet OWASP guidelines.

Fix Comment

As we recommended, the project team modified the code.

17. Data Inconsistency Caused by Deleted and Re-created Ledger Entries

ID: RIGO-17
Type: Logic Error
File: ledger/finality_ledger.go

Severity: Low
Difficulty: Medium

Issue

When a key is deleted from the ledger, the information about the keys to be removed is recorded in the removedKeys data structure, which is then deleted once the block is committed.

However, even if data is deleted and subsequently recreated in the ledger, the information stored in removedKeys is not cleared.

As a result, if data is deleted and recreated within a single block, the newly created data will be deleted at the commit point due to the corresponding entry in removedKeys.

```
// rigo-go/ledger/finality_ledger.go#L92
func (ledger *FinalityLedger[T]) DelFinality(key LedgerKey) (T, xerrors.XError) {
	ledger.mtx.Lock()
	defer ledger.mtx.Unlock()

	// in case of deleting,
	// it should be removed from SimpleLedger too.
	_, _ = ledger.SimpleLedger.del(key)

	var emptyNil T

	if item, err := ledger.getFinality(key); err != nil {
		return emptyNil, err
	} else {
		ledger.finalityItems.delGotItem(key)       // delete(ledger.gotItems, key)
		ledger.finalityItems.delUpdatedItem(key)   // delete(ledger.updatedItems,
key)
		ledger.finalityItems.appendRemovedKey(key) // ledger.removedKeys =
append(ledger.removedKeys, key)

		return item, nil
	}
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ledger/finality_ledger.go#L92-L111

```
// rigo-go/ledger/finality_ledger.go#L139
```

```
func (ledger *FinalityLedger[T]) Commit() ([]byte, int64, xerrors.XError) {
	ledger.mtx.Lock()
	defer ledger.mtx.Unlock()

	// remove
	for _, k := range ledger.finalityItems.removedKeys {
		var vk LedgerKey
		copy(vk[:], k[:])
		if _, _, err := ledger.tree.Remove(vk[:]); err != nil {
			return nil, -1, xerrors.From(err)
		}
		delete(ledger.finalityItems.gotItems, vk)
		delete(ledger.finalityItems.updatedItems, vk)

		// this item should be removed from SimpleLedger too.
		delete(ledger.SimpleLedger.cachedItems.gotItems, vk)
		delete(ledger.SimpleLedger.cachedItems.updatedItems, vk)
	}
	...
}
```

https://github.com/rigochain/rigo-go/blob/31e98792ea106f9cfa54952a59ec328b3bd40936/ledger/finality_ledger.go#L139-L182

Recommendation

We recommend removing the corresponding key from the "removedKeys" list if the key created when setting a new value in the ledger is included in it.

Fix Comment

As we recommended, the project team modified the code.

18. Insufficient Minimum Staking Requirement for Validators

ID: RIGO-18
Type: Logic Error
File: -

Severity: High
Difficulty: Medium

Issue

The absence of a minimum staking requirement allows users with a low staked amount to become validators, potentially leading to malicious behavior, disruption of the consensus algorithm, and increased vulnerability to slashing penalties.

The current implementation has no minimum staking requirement for users to become validators.

As a result, if the maxValidatorCnt value is set high or if a validator unstakes, creating a vacant validator slot, users with a relatively low staked amount can become validators.

This allows malicious actors to disrupt the consensus algorithm or propose malicious proposals, as the penalties for slashing are relatively low for validators with low stakes.

Recommendation

We recommend implementing a reasonable minimum staking requirement for users to become validators, ensuring that only users with a significant stake can participate as validators.

Fix Comment

As we recommended, the project team modified the code.

19. Insufficient Token Distribution Explanation and Misalignment with Block Rewards-Based Distribution Design

ID: RIGO-19
Type: Logic Error
File: ctrlers/types/block_ctx.go

Severity: Tips
Difficulty: -

Issue

Token circulation deviates from the team's planned token distribution outlined in the documentation.

Currently, the distribution plan of the project does not include information about the native token generated as block rewards.

Additionally, there are no limitations in place for the block rewards issued. Validators' block rewards increase proportionally to the number of native tokens staked, but without any restrictions on the block rewards, there is a possibility that the token circulation may deviate from the team's planned distribution model or tokenomics.

Recommendation

We recommend defining and documenting the native token information related to block rewards in the token economic documentation.

This should include details such as the issuance rate, allocation mechanism, and distribution plan.

Additionally, it is important to implement limitations on block rewards to align with the intended token distribution plan and prevent potential discrepancies.

Fix Comment

The Ligo token economics document has been updated to reflect this information.

DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the main network. In order to write secure codes, correction of discovered problems and sufficient testing thereof are required.

Appendix. A

Severity Level

CRITICAL	Must be addressed as a vulnerability that has the potential to seize or freeze substantial sums of money.
HIGH	Has to be fixed since it has the potential to deny users compensation or momentarily freeze assets.
MEDIUM	Vulnerabilities that could halt services, such as DoS and Out-of-Gas, need to be addressed.
LOW	Issues that do not comply with standards or return incorrect values
TIPS	Tips that makes the code more usable or efficient when modified

Difficulty Level

	Low	Medium	High
Privilege	anyone	Miner/Block Proposer	Admin/Owner
Capital needed	Small or none	Gas fee or volatile as price change	More than exploited amount
Probability	100%	Depend on environment	Hard as mining difficulty

Vulnerability Category

Arithmetic	• Integer under/overflow vulnerability • floating point and rounding accuracy
Access & Privilege Control	• Manager functions for emergency handle • Crucial function and data access • Count of calling important task, contract state change, intentional task delay
Denial of Service	• Unexpected revert handling • Gas limit excess due to unpredictable implementation
Miner Manipulation	• Dependency on the block number or timestamp. • Frontrunning
Reentrancy	•Proper use of Check-Effect-Interact pattern. •Prevention of state change after external call • Error handling and logging.
Low-level Call	• Code injection using delegatecall • Inappropriate use of assembly code
Off-standard	• Deviate from standards that can be an obstacle of interoperability.

Input Validation	• Lack of validation on inputs.
Logic Error/Bug	• Unintended execution leads to error.
Documentation	•Coherency between the documented spec and implementation
Visibility	• Variable and function visibility setting
Incorrect Interface	• Contract interface is properly implemented on code.

End of Document