SOFTWARE AUDIT REPORT

for

POC CHAIN

Prepared By: Shuxiao Wang

PeckShield

Feb. 05, 2021

Document Properties

Client	POC Chain
Title	Software Audit Report
Target	POC
Version	1.0
Author	Ruiyi Zhang
Auditors	Ruiyi Zhang, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	Feb. 05, 2021	Ruiyi Zhang	Final Release
1.0-rc	Feb. 04, 2021	Ruiyi Zhang	Release Candidate
0.2	Jan. 28, 2021	Ruiyi Zhang	Additional Findings
0.1	Jan. 20, 2021	Ruiyi Zhang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **POC** design document and related source code, we outline in this report our systematic method to evaluate potential security issues in the POC implementation, expose possible semantic inconsistencies between the source code and the design specification, and provide additional suggestions and recommendations for improvement. Our results show that the given branch of POC can be further improved due to the presence of several issues related to either security or performance. This document describes our audit results in detail.

1.1 About POC

The POC blockchain provides an efficient distributed on-chain identity, storage, and transaction system, supports smart contracts and virtual machines in multiple languages, provides two modules of core protocols and application frameworks, and guarantees data security through a consensus mechanism to achieve a complete centralized blockchain system. The core protocol provides a POC token framework. Around the economic model of proof of contribution value, the cross-chain protocol is used to implement the layer-to-layer and chain-to-chain interaction protocols. The application framework combines the smart contracts on the chain and the off-chain front-end to interact and provides targeted solutions to the chain reform needs of different industries in the real economy.

The basic information of POC is as follows:

Table 1.1: Basic Information of POC

Item	Description
Issuer	POC Chain
Website	https://www.pocblockchain.io/
Type	POC Chain
Platform	Go#
Audit Method	White-box
Latest Audit Report	Feb. 05, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit:

- https://github.com/pocblockchain/pocc (686cbed)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/pocblockchain/pocc (2d3f891)

1.2 About PeckShield

PeckShield Inc. [1] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products including security audits. We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

In the first phase of auditing POC, we use fuzzing to find out the corner cases that may not be covered by in-house testing. Next we do white-box auditing, in which PeckShield security auditors manually review POC design and source code, analyze them for any potential issues, and follow up with issues found in the fuzzing phase. If necessary, we design and implement individual test cases to further reproduce and verify the issues. In the following subsections, we will introduce the risk model as well as the audit procedure adopted in this report.

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3.1 Risk Model

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [2]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, and *Low* shown in Table 1.2.

1.3.2 Fuzzing

Fuzzing or fuzz testing is an automated software testing technique of discovering software vulnerabilities by systematically finding and providing possible inputs to the target program, and then monitoring the program execution for crashes (or any unexpected results). In the first phase of our audit, we use fuzzing to find out possible corner cases or unusual inter-module interactions that may not be covered by in-house testing. As one of the most effective methods for exposing the presence of possible vulnerabilities, fuzzing technology has been the first choice for many security researchers in recent years. At present, there are many fuzzy testing tools and supporting software, which can help security personnels to conduct fuzzing and find vulnerabilities more efficiently. Based on the characteristics of the POC, we use AFL [3] as the primary tool for fuzz testing.

AFL (American Fuzzy Lop) is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. Since its inception, AFL has gained growing popularity in the industry and has proved its effectiveness in discovering quite a few significant software bugs in a wide range of major software projects. The basic process of AFL fuzzing is as follows:

- Generate compile-time instrumentation to record information such as code execution path;
- Construct some input files to join the input queue, and change input files according to different strategies;
- Files that trigger a crash or timeout when executing an input file are logged for subsequent analysis;

- Loop through the above process.

Throughout the AFL testing, we will reproduce each crash based on the crash file generated by AFL. For each reported crash case, we will further analyze the root cause and check whether it is indeed a vulnerability. Once a crash case is confirmed as a vulnerability of the POC, we will further analyze it as part of the white-box audit.

1.3.3 White-box Audit

After fuzzing, we continue the white-box audit by manually analyzing source code. Here we test target software's internal structure, design, coding, and we focus on verifying the flow of input and output through the application as well as examining possible design and implementation trade-offs for strengthened security. PeckShield auditors first fully review and understand the source code, then create specific test cases, execute them and analyze the results. Issues such as internal security loopholes, unexpected output, broken or poorly structured paths, etc., will be inspected under close scrutiny.

Blockchain is a secure method of creating a distributed database of transactions, and three major technologies of blockchain are cryptography, decentralization, and consensus model. Blockchain does come with unique security challenges, and based on our understanding of blockchain general design, we in this audit divide the blockchain software into the following major areas and inspect each area accordingly:

- Data and state storage, which is related to the database and files where blockchain data are saved.
- P2P networking, consensus, and transaction model in the networking layer. Note that the consensus and transaction logic is tightly coupled with networking.
- VM, account model, and incentive model. This is essentially the execution and business layer of the blockchain, and many blockchain business specific logics are implemented here.
- System contracts and services. These are system-level, blockchain-wide operation management contracts and services.
- Others. This includes any software modules that do not belong to above-mentioned areas, such as common crypto or other 3rd-party libraries, best practice or optimization used in other software projects, design and coding consistency, etc.

Based on the above classification, we show in Table 1.3 and Table 1.4 the detailed list of the audited items in this report.

Table 1.3: The Full List of Audited Items (Part I)

Category	Check Item
Data and State Storage	Blockchain Database Security
	Database State Integrity Check
Node Operation	Default Configuration Security
	Default Configuration Optimization
	Node Upgrade And Rollback Mechanism
Node Communication	External RPC Implementation Logic
	External RPC Function Security
	Node P2P Protocol Implementation Logic
	Node P2P Protocol Security
	Serialization/Deserialization
	Invalid/Malicious Node Management Mechanism
	Communication Encryption/Decryption
	Eclipse Attack Protection
	Fingerprint Attack Protection
Consensus	Consensus Algorithm Scalability
	Consensus Algorithm Implementation Logic
	Consensus Algorithm Security
Transaction Model	Transaction Privacy Security
	Transaction Fee Mechanism Security
	Transaction Congestion Attack Protection
VM	VM Implementation Logic
	VM Implementation Security
	VM Sandbox Escape
	VM Stack/Heap Overflow
	Contract Privilege Control
	Predefined Function Security
Account Model	Status Storage Algorithm Adjustability
	Status Storage Algorithm Security
	Double Spending Protection
Incentive Model	Mining Algorithm Security
	Mining Algorithm ASIC Resistance
	Tokenization Reward Mechanism

Table 1.4: The Full List of Audited Items (Part II)

Category	Check Item
System Contracts And Services	Memory Leak Detection
	Use-After-Free
	Null Pointer Dereference
	Undefined Behaviors
	Deprecated API Usage
	Signature Algorithm Security
	Multisignature Algorithm Security
SDK Security	Using RPC Functions Security
	Privatekey Algorithm Security
	Communication Security
	Function integrity checking code
Others	Third Party Library Security
	Memory Leak Detection
	Exception Handling
	Log Security
	Coding Suggestion And Optimization
	White Paper And Code Implementation Uniformity

To better describe each issue we identified, we also categorize the findings based on Common Weakness Enumeration (CWE-699) [4], which is a community-developed list of software weakness types to better classify and organize weaknesses around concepts frequently encountered in software development. We use the CWE categories in Table 1.5 to classify our findings.

Table 1.5: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Input Validation Issues	Weaknesses in this category are related to a software system's input validation components. Frequently these deal with sanitizing, neutralizing and validating any externally provided inputs to minimize malformed data from entering the system and preventing code injection in the input data.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of blockchain software. Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the POC implementation. During the first phase of our audit, we study the source code and ran our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tools. After that, we manually review business logics, examine system operations, and place operation-specific aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	5	■ ■ ■ ■ ■
Informational	4	■ ■ ■ ■
Total	10	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, the POC are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 5 low-severity vulnerabilities, and 4 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Calculation of Validator's Voting Power	Business Logic	Resolved
PVE-002	Low	Tally Calculation Precision Error	Numeric Errors	Resolved
PVE-003	Low	Inappropriate Logic of Unjail of Non-Bonded Jailed Validator	Business Logic	Resolved
PVE-004	Low	Incomplete Genesis State in Staking Module	Init. & Cleanup	Resolved
PVE-005	Low	Missed Amount Event in InputOutputCoins()	Coding Practices	Resolved
PVE-006	Informational	Excessive Sanity Checks in handleMsgMulti-Send()	Coding Practices	Resolved
PVE-007	Low	Incorrect Implementation of Basic Data Types	Business Logic	Resolved
PVE-008	Informational	Excessive Sanity Checks in handleMsgInflate-Token()	Coding Practices	Resolved
PVE-009	Informational	Erroneous Description in Mint Module	Coding Practices	Resolved
PVE-010	Informational	Potentially Halted Chain By The Low Inflation Rate	Init. & Cleanup	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Chapter 3 for details.

3 Detailed Results

3.1 Incorrect Calculation of Validator's Voting Power

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `x/gov/tally.go`
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [6]

Description

POC is developed on version 0.37.2 of Cosmos-SDK[7], a popular modular framework for building application-specific blockchains. Note that Cosmos-SDK enables rapid development of SDK-based blockchains out of composable modules.

The modular Cosmos-SDK framework allows various modules to generally handle a subset of the state and, as such, each module can be seen as a little state-machine. In this section, we introduce an issue found in `gov` module. The `gov` module enables Cosmos-SDK based blockchain to support an on-chain governance system. In this system, holders of the native staking token of the chain can vote on proposals. According to its spec described, if the delegator votes before its validator, it will not inherit from the validator's vote. However, the `tally` function was incorrectly handling validators who delegated to other validators. Specifically, we show below the related code snippet inside the `tally()` handler. If an address is a validator operator and it's delegated to another validator. When the address vote for a proposal, it just sums the voting power about its own validator (line 61).

```
54   keeper.IterateVotes(ctx, proposal.ProposalID, func(vote types.Vote) bool {
55     // if validator, just record it in the map
56     // if delegator tally voting power
57     valAddrStr := sdk.ValAddress(vote.Voter).String()
58     if val, ok := currValidators[valAddrStr]; ok {
59       val.Vote = vote.Option
60       currValidators[valAddrStr] = val
61     } else {
```

```
        // iterate over all delegations from voter, deduct from any delegated-to
            validators
        keeper.sk.IterateDelegations(ctx, vote.Voter, func(index int64, delegation
            exported.DelegationI) (stop bool) {
          valAddrStr := delegation.GetValidatorAddr().String()

          if val, ok := currValidators[valAddrStr]; ok {
            val.DelegatorDeductions = val.DelegatorDeductions.Add(delegation.GetShares())
            currValidators[valAddrStr] = val

            delegatorShare := delegation.GetShares().Quo(val.DelegatorShares)
            votingPower := delegatorShare.MulInt(val.BondedTokens)

            results[vote.Option] = results[vote.Option].Add(votingPower)
            totalVotingPower = totalVotingPower.Add(votingPower)
          }

          return false
        })
      }

      keeper.deleteVote(ctx, vote.ProposalID, vote.Voter)
      return false
    })
```

Listing 3.1: poc/x/gov/tally.go

Recommendation Appropriately revise the `tally()` logic as follow:

```
keeper.IterateVotes(ctx, proposal.ProposalId, func(vote types.Vote) bool {
    // if validator, just record it in the map
    voter, err := sdk.AccAddressFromBech32(vote.Voter)

    if err != nil {
        panic(err)
    }

    valAddrStr := sdk.ValAddress(voter.Bytes()).String()
    if val, ok := currValidators[valAddrStr]; ok {
        val.Vote = vote.Option
        currValidators[valAddrStr] = val
    }

    // iterate over all delegations from voter, deduct from any delegated-to validators
    keeper.sk.IterateDelegations(ctx, voter, func(index int64, delegation stakingtypes.
        DelegationI) (stop bool) {
        valAddrStr := delegation.GetValidatorAddr().String()

        if val, ok := currValidators[valAddrStr]; ok {
            // There is no need to handle the special case that validator address equal
                to voter address.
            // Because voter's voting power will tally again even if there will deduct
                voter's voting power from validator.
```

```
            val.DelegatorDeductions = val.DelegatorDeductions.Add(delegation.GetShares()
                )
            currValidators[valAddrStr] = val

            // delegation shares * bonded / total shares
            votingPower := delegation.GetShares().MulInt(val.BondedTokens).Quo(val.
                DelegatorShares)

            results[vote.Option] = results[vote.Option].Add(votingPower)
            totalVotingPower = totalVotingPower.Add(votingPower)
        }

        return false
    })

    keeper.deleteVote(ctx, vote.ProposalId, voter)
    return false
})
```

Listing 3.2: poc/x/gov/tally.go

Status The issue has been fixed by this commit: `2d3f891`

3.2 Tally Calculation Precision Error

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `x/gov/tally.go`
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [9]

Description

As described in Section 3.1, the `gov` module supports four features related to on-chain governance system: i.e., `Proposal submission`, `Vote`, `Inheritance and penalties`, and `Claiming deposit`.

In the governance system, the voting power metric is representative of how much a stake a validator has been allocated by delegators and thus determines how likely they are to be selected by the network to vote on and propose a new block.

When handling the `queryTally` message type, there is a need to calculate the voting power. However, it is currently calculated in way that may lead to precision loss (line 71): `delegation.GetShares().Quo(val.DelegatorShares).MulInt(val.BondedTokens)`.

```
62    // iterate over all delegations from voter, deduct from any delegated-to validators
63    keeper.sk.IterateDelegations(ctx, vote.Voter, func(index int64, delegation exported.
        DelegationI) (stop bool) {
```

```
        valAddrStr := delegation.GetValidatorAddr().String()

        if val, ok := currValidators[valAddrStr]; ok {
            val.DelegatorDeductions = val.DelegatorDeductions.Add(delegation.GetShares())
            currValidators[valAddrStr] = val

            delegatorShare := delegation.GetShares().Quo(val.DelegatorShares)
            votingPower := delegatorShare.MulInt(val.BondedTokens)

            results[vote.Option] = results[vote.Option].Add(votingPower)
            totalVotingPower = totalVotingPower.Add(votingPower)
        }

        return false
    })
```

Listing 3.3: poc/x/gov/tally.go

```
    // iterate over the validators again to tally their voting power
  for _, val := range currValidators {
    if val.Vote == OptionEmpty {
      continue
    }

    sharesAfterDeductions := val.DelegatorShares.Sub(val.DelegatorDeductions)
    fractionAfterDeductions := sharesAfterDeductions.Quo(val.DelegatorShares)
    votingPower := fractionAfterDeductions.MulInt(val.BondedTokens)

    results[val.Vote] = results[val.Vote].Add(votingPower)
    totalVotingPower = totalVotingPower.Add(votingPower)
  }
```

Listing 3.4: poc/x/gov/tally.go

For improved precision, it is suggested to calculate the multiplication before the division, i.e., `votingPower := delegation.GetShares().MulInt(val.BondedTokens).Quo(val.DelegatorShares)`.

Recommendation Revise the above calculation to better handle possible precision loss: i.e., `votingPower := delegation.GetShares().MulInt(val.BondedTokens).Quo(val.DelegatorShares)`.

Status The issue has been fixed by this commit: `2d3f891`

3.3 Inappropriate Logic of Unjail of Non-Bonded Jailed Validator

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `x/slashing/handler.go`
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [10]

Description

Each block, the top `MaxValidators` (defined by `x/staking`) validators who are not jailed become bonded, meaning that they may propose and vote on blocks. Validators who are bonded are at stake, meaning that part or all of their stake and their delegators' stake is at risk if they commit a protocol fault. For each of these validators POC keeps a `ValidatorSigningInfo` record that contains information pertaining to the validator's liveness and other infraction related attributes.

Validators must submit a transaction to `unjail` itself after having been jailed (and thus unbonded) for downtime. Specifically, we show below the related code snippet inside the `handleMsgUnjail()` handler. If the validator has a `ValidatorSigningInfo` object that signals that the validator was bonded and so we must check that the validator is not tombstoned and can be unjailed at the current block . However, a validator that is jailed but has no ValidatorSigningInfo object signals that the validator was never bonded and must've been jailed due to falling below their minimum self-delegation. The validator should be unjailed at any point assuming they've now bonded above their minimum self-delegation.

```go
	info, found := k.getValidatorSigningInfo(ctx, consAddr)
	if !found {
		return ErrNoValidatorForAddress(k.codespace).Result()
	}

	// cannot be unjailed if tombstoned
	if info.Tombstoned {
		return ErrValidatorJailed(k.codespace).Result()
	}

	// cannot be unjailed until out of jail
	if ctx.BlockHeader().Time.Before(info.JailedUntil) {
		return ErrValidatorJailed(k.codespace).Result()
	}

	k.sk.Unjail(ctx, consAddr)
```

Listing 3.5: poc/x/slashing/handler.go

Recommendation Allow a validator to immediately unjail when no signing info is present due to falling below their minimum self-delegation and never having been bonded. The validator may immediately unjail once they've met their minimum self-delegation.

```go
    if found {
        // cannot be unjailed if tombstoned
        if info.Tombstoned {
            return types.ErrValidatorJailed
        }

        // cannot be unjailed until out of jail
        if ctx.BlockHeader().Time.Before(info.JailedUntil) {
            return types.ErrValidatorJailed
        }
    }
```

Listing 3.6: poc/x/slashing/handler.go

Status The issue has been fixed by this commit: `2d3f891`

3.4 Incomplete Genesis State in Staking Module

- ID: PVE-004
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `x/staking/genesis.go`
- Category: Initialization & Cleanup [11]
- CWE subcategory: CWE-459 [12]

Description

As described in Section 3.1, the modular Cosmos-SDK framework allows various modules to generally handle a subset of the state and, as such, these modules need to define the related subset of the genesis file as well as methods to initialize, verify, and export it. We stress that these states are essential to the blockchain's genesis state import and export and are therefore required for seamless upgrades. In the current POC codebase, `staking` module does not have thorough genesis-related states properly exported. Specifically, the RPC for the genesis contains the validator's address, but the validator's address is not exported in the `WriteValidators` logic. This discrepancy may cause clients to fail, since they don't calculate the same GCI hash as for the exported `genesis.json`.

```go
    // WriteValidators returns a slice of bonded genesis validators.
    func WriteValidators(ctx sdk.Context, keeper Keeper) (vals []tmtypes.
        GenesisValidator) {
        keeper.IterateLastValidators(ctx, func(_ int64, validator exported.ValidatorI) (
            stop bool) {
            vals = append(vals, tmtypes.GenesisValidator{
```

```
                PubKey:  validator.GetConsPubKey(),
                Power:   validator.GetConsensusPower(),
                Name:    validator.GetMoniker(),
            })

            return false
        })

        return
    }
```

Listing 3.7: poc/x/staking/genesis.go

Similar genesis-related issues are also present in `genutil` module. Fix `ExportGenesis` as to not return null, but instead of default genesis state ([]), so on export, genesis validation should pass.

Recommendation Appropriately export necessary genesis state in affected modules

```
    // WriteValidators returns a slice of bonded genesis validators.
    func WriteValidators(ctx sdk.Context, keeper Keeper) (vals []tmtypes.
        GenesisValidator) {
        keeper.IterateLastValidators(ctx, func(_ int64, validator exported.ValidatorI) (
            stop bool) {
            vals = append(vals, tmtypes.GenesisValidator{
                Address: validator.GetConsAddr().Bytes(),
                PubKey:  validator.GetConsPubKey(),
                Power:   validator.GetConsensusPower(),
                Name:    validator.GetMoniker(),
            })

            return false
        })

        return
    }
```

Listing 3.8: poc/x/staking/genesis.go

```
func (am AppModule) ExportGenesis(ctx sdk.Context, cdc codec.JSONMarshaler) json.
    RawMessage {
  return am.DefaultGenesis(cdc)
}
```

Listing 3.9: poc/x/genutil/module.go

Status The issue has been fixed by this commit: `2d3f891`

3.5 Missed Amount Event in InputOutputCoins()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `x/bank/internal/keeper/keeper.go`
- Category: Coding Practices [13]
- CWE subcategory: CWE-1116 [14]

Description

Among all modules in POC, the `bank` module is one of the most crucial modules and its main functionality is to handle multi-asset coin transfer between accounts. Its complexity is also partially reflected in the number of messages it recognizes and handles. In total, there are 6 different types of messages.

In this section, we mainly focus on the three specific message types related to multi-accounts' asset transfers: i.e., `MsgMultiSend`, `MsgBonusSend` and `MsgReclaimSend`. In the code routines for handling these three message types, the function `InputOutputCoins` will be called to transfers coins from any number of input accounts to any number of output accounts. However, this particular function misses amount event for each output (lines 214-219). There is a discrepancy between `SendCoins` and `InputOutputCoins`.

```
208     for _, out := range outputs {
209     _, err := keeper.AddCoins(ctx, out.Address, out.Coins)
210     if err != nil {
211       return err
212     }
213
214     ctx.EventManager().EmitEvent(
215       sdk.NewEvent(
216         types.EventTypeTransfer,
217         sdk.NewAttribute(types.AttributeKeyRecipient, out.Address.String()),
218       ),
219     )
220   }
```

Listing 3.10: poc/x/bank/internal/keeper/keeper.go

Recommendation Add amount event to InputOutputCoins for `MsgMultiSend`, `MsgBonusSend` and `MsgReclaimSend`.

Status The issue has been fixed by this commit: `2d3f891`

3.6 Excessive Sanity Checks in handleMsgMultiSend()

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `x/bank/handler.go`
- Category: Coding Practices [13]
- CWE subcategory: CWE-1050 [15]

Description

As described in Section 3.5, the `token` module supports three specific message types related to multi-accouts' asset transfers: i.e., `MsgMultiSend`, `MsgBonusSend` and `MsgReclaimSend`. Using the message type `MsgMultiSend` as an example, we show below the implementation of current `handleMsgMultiSend()` routine. As the comments described, `totalIn == totalOut` should have been checked. Consequently, there is an excessive sanity check that ensures the coins of `msg.Outputs` are all `SendEnabled` (line 88).

```
70    // Handle MsgMultiSend.
71    func handleMsgMultiSend(ctx sdk.Context, k keeper.Keeper, msg types.MsgMultiSend)
          sdk.Result {
72        // NOTE: totalIn == totalOut should already have been checked
73        if !k.GetSendEnabled(ctx) {
74            return types.ErrSendDisabled(k.Codespace()).Result()
75        }
76
77        for _, in := range msg.Inputs {
78            if !k.IsCoinsSendEnabled(ctx, in.Coins) {
79                return types.ErrSendDisabled(k.Codespace()).Result()
80            }
81        }
82
83        for _, out := range msg.Outputs {
84            if k.BlacklistedAddr(out.Address) {
85                return sdk.ErrUnauthorized(fmt.Sprintf("%s is not allowed to receive
                      transactions", out.Address)).Result()
86            }
87
88            if !k.IsCoinsSendEnabled(ctx, out.Coins) {
89                return types.ErrSendDisabled(k.Codespace()).Result()
90            }
91        }
```

Listing 3.11: poc/x/bank/handler.go

Similar duplicate-checks issues are also present in the `handleMsgBonusSend()` and `handleMsgReclaimSend()` routines.

Recommendation Remove the duplicate check in the function `handleMsgInflateToken()`

Status The issue has been fixed by this commit: `2d3f891`

3.7 Incorrect Implementation of Basic Data Types

- ID: PVE-007
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `types`
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [10]

Description

In POC, the `type` package defines a series of basic data types, E.g. `Int`, `Uint`, and `Decimal`. Specifically, `Int` and `Uint` both wrap integer with 256 bit range bound Checks overflow, underflow and division by zero. `Int` exists in range from `-(2^maxBitLen-1)` to `2^maxBitLen-1`, and `Uint` exists in range from `0` to `2^256-1`. During this audit process, we stress two issues related to the two types as below:

- Unsigned Integer Less-Than-Equal incorrectly evaluates when comparing equal numbers (line 72).
- Panic when calling `BigInt()` on an uninitialized `Int` (lines 107-109).

```
71  // LTE returns true if first Uint is lesser than or equal to the second
72  func (u Uint) LTE(u2 Uint) bool { return !u.GTE(u2) }
```

Listing 3.12: poc/types/uint.go

```
107 // BigInt converts Int to big.Int
108 func (i Int) BigInt() *big.Int {
109     return new(big.Int).Set(i.i)
110 }
```

Listing 3.13: poc/types/int.go

Recommendation Correct the corresponding code as spec.

```
71  // LTE returns true if first Uint is lesser than or equal to the second
72  func (u Uint) LTE(u2 Uint) bool { return !u.GT(u2) }
```

Listing 3.14: poc/types/uint.go

```
107 // BigInt converts Int to big.Int
108 func (i Int) BigInt() *big.Int {
109     if i.IsNil() {
110     return nil
111   }
```

```
        return new(big.Int).Set(i.i)
    }

    // IsNil returns true if Int is uninitialized
    func (i Int) IsNil() bool {
        return i.i == nil
    }
```

Listing 3.15: poc/types/int.go

Status The issue has been fixed by this commit: `2d3f891`

3.8 Excessive Sanity Checks in handleMsgInflateToken()

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `token`
- Category: Coding Practices [13]
- CWE subcategory: CWE-1050 [15]

Description

In this section, we examine the `token` module and find an informational issue. The `token` module enlists available tokens for trading and supports three features, i.e., `handleMsgNewToken()`, `handleMsgInflateToken()` and `handleMsgBurnToken()`. There is an excessive check on `handleMsgInflateToken()` that can be deleted to make the code more concise and efficient. Specifically, the function `ValidateBasic` of `MsgInflateToken` has ensured that `len(msg.Amount)== 1`, but the length of inflated coins is checked again in `handleMsgInflateToken()`.

```
    // ValidateBasic runs stateless checks on the message
    func (msg MsgInflateToken) ValidateBasic() sdk.Error {
        if msg.From.Empty() {
            return sdk.ErrInvalidAddress(fmt.Sprintf("from address can not be empty:%v",
                msg.From))
        }

        if msg.To.Empty() {
            return sdk.ErrInvalidAddress(fmt.Sprintf("to address can not be empty%v",
                msg.To))
        }

        if len(msg.Amount) != 1 {
            return sdk.ErrInvalidTx(fmt.Sprintf("inflate only ONE coin once"))
        }
```

Listing 3.16: poc/x/token/types/msgs.go

```
func handleMsgInflateToken(ctx sdk.Context, keeper Keeper, msg types.MsgInflateToken
    ) sdk.Result {
    ctx.Logger().Info("handleMsgInflateToken", "msg", msg)

    if len(msg.Amount) != 1 {
        return sdk.ErrInvalidTx(fmt.Sprintf("inflate only ONE coin once")).Result()
    }
```

Listing 3.17: poc/x/token/handler.go

Recommendation Remove the duplicate checks in the function `handleMsgInflateToken()`

Status The issue has been fixed by this commit: `2d3f891`

3.9 Erroneous Description in Mint Module

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Mint`
- Category: Coding Practices [13]
- CWE subcategory: CWE-1116 [14]

Description

In this section, we examine the minting logic in POC. Minting parameters are recalculated and inflation paid at the beginning of each block. The function `BlockProvision()` calculates the provisions generated for each block based on current annual provisions. The provisions are then minted by the mint module's ModuleMinterAccount and then transferred to the auth's FeeCollector ModuleAccount. In the current implementation of POC, `AnnualProvisions` is calculated by `InitalInflationAmount` and `DefaultInflationFactor`. To elaborate, we show below the logic of `AnnualProvisions` calculation.

```
//AnnualProvisions in current year = AnnualProvisions in previous year * InflationFactor
/*
   year       AnnualProvisions
    1          81000000 *InflationFactor[0]
    2          81000000 *InflationFactor[0] * InflationFactor[1]
    3          81000000 *InflationFactor[0] * InflationFactor[1] * InflationFactor[2]
    4          81000000 *InflationFactor[0] * InflationFactor[1] * InflationFactor[2] *
        InflationFactor[3]
    5          81000000 *InflationFactor[0] * InflationFactor[1] * InflationFactor[2] *
        InflationFactor[3] * InflationFactor[4]
    6          81000000 *InflationFactor[0] * InflationFactor[1] * InflationFactor[2] *
        InflationFactor[3] * pow(InflationFactor[4],2)
    .... ....
    n          81000000 *InflationFactor[0] * InflationFactor[1] * InflationFactor[2] *
        InflationFactor[3] * pow(InflationFactor[4], n-4)
```

```
*/
```

Listing 3.18: poc/x/mint/internal/types/minter.go

However, the description of the annual inflation is inconsistent with the code logic (line 44).

```
43     // DefaultInitialMinter returns a default initial Minter object for a new chain
44     // which uses an inflation rate of 13%.
45     func DefaultInitialMinter(annualProvisions sdk.Dec) Minter {
46         return InitialMinter(0, annualProvisions)
47     }
```

Listing 3.19: poc/x/mint/internal/types/minter.go

Recommendation Correct the corresponding descriptions.

Status The issue has been fixed by this commit: `2d3f891`

3.10 Potentially Halted Chain By The Low Inflation Rate

- ID: PVE-010
- Severity: Informational
- Likelihood: Medium
- Impact: N/A
- Target: `Mint`
- Category: Initialization & Cleanup [11]
- CWE subcategory: CWE-1188 [16]

Description

In `cosmos-sdk` v0.37.2 version, if a chain starts with a low inflation rate (or the low number of coins), an invalid mint module account is created. This invalid account can then trigger a chain halt if the inflation rate (or coin amount) is increased. Specifically, for new chains, after `InitGenesis()`, mint's module account doesn't exist. Normaly it's created as needed when mint's `beginblocker()` runs. Mint's begin blocker calls `supply.Mint()` which creates a mod account if one doesn't exist. Then newly minted coins are sent from this to the fee mod account.

However, if the coins to be minted in that initial block are 0, then minting is skipped, so no account is created. But sending coins from this non-existent account to the fee mod account is not skipped, which triggers the bank module to create an empty `BaseAccount` with the address for mint's module account. Finally, if `supply` ever accesses this account it panics in the type assertion trying to convert it to a module account.

```
macc, ok := acc.(exported.ModuleAccountI)
```

Listing 3.20: cosmos-sdk/x/supply/internal/keeper/account.go

It is worth mentioning that in the current implementation of POC, this problem does not exist because the amount of inflated token is large enough. But our suggestion is still to fix this issue to avoid unnecessary risks.

Recommendation Create module Account for mint module on `InitGenesis`.

Status The issue has been fixed by this commit: `2d3f891`

3.11 Other Suggestions

As a common suggestion, due to the fact that POC is developed on version 0.32.6 of Tendermint and Tendermint is continunously updated, it is always preferred to use fixed Tendermint versions whenever possible. We highly encourage to use a relatively new version Tendermint, e.g., 0.34.3 instead of 0.32.6, since Tendermint has patched several new security bugs in the intermediate updates.

4 Conclusion

In this security audit, we have analyzed the POC design and implementation. The core protocol provides a POC token framework. Around the economic model of proof of contribution value, the cross-chain protocol is used to implement the layer-to-layer and chain-to-chain interaction protocols. Our audit has uncovered a list of 10 potential issues, and some of them involve unusual interactions among multiple modules.

Our journey through this audit is that the POC software is neatly organized and elegantly implemented and those identified issues are promptly confirmed and fixed. We would ike to commend the team for a well-done software project, and for quickly fixing reported issues. Also, as expressed in Section 1.4, we appreciate any constructive feedback or suggestions about this report.

References

[1] PeckShield. PeckShield Inc. https://www.peckshield.com.

[2] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[3] Lcamtuf. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[4] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[5] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[7] Tendermint Inc. Cosmos SDK Documentation. https://docs.cosmos.network/.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[10] MITRE. CWE: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[11] MITRE. CWE-452: Initialization and Cleanup Errors. https://cwe.mitre.org/data/definitions/452.html.

[12] MITRE. CWE-459: Incomplete Cleanup. https://cwe.mitre.org/data/definitions/459.html.

[13] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[14] MITRE. CWE: Inaccurate Comments. https://cwe.mitre.org/data/definitions/1116.html.

[15] MITRE. CWE CATEGORY: Excessive Platform Resource Consumption within a Loop. https://cwe.mitre.org/data/definitions/1050.html.

[16] MITRE. CWE-1188: Insecure Default Initialization of Resource. https://cwe.mitre.org/data/definitions/1188.html.