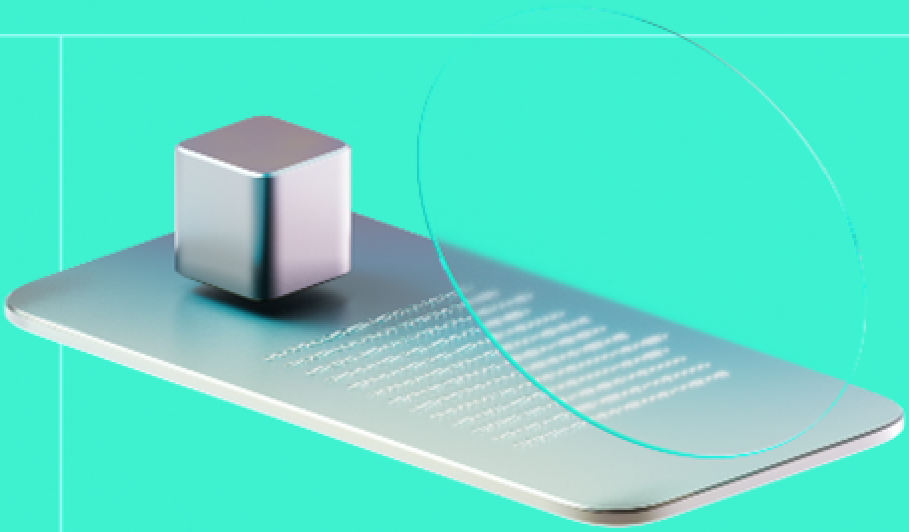
Blockchain Protocol Security Analysis Report



Customer: Areon

Date: 23/01/2024

We express our gratitude to the Areon team for the collaborative engagement that enabled the execution of this Security Assessment.

Areon is a decentralized layer 1 blockchain meant for daily usage with low fees and high transaction throughput. It also allows anyone to build dApps on top of its architecture.

Platform: AreonChain

Language: Go

Timeline: 29/11/2023 - 19/01/2024

Methodology: [Blockchain Protocol and Security Analysis Methodology]

Review Scope

Repository	https://github.com/Areon-Network/AreonChain
Commit	acf467a

Audit Summary

10/10	8/10	10/10	9/10
Security Score	Code quality score	Architecture quality score	Documentation quality score

Total 9/10

The system users should acknowledge all the risks summed up in the risks section of the report

4	4	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	0
Medium	1
Low	3

Vulnerability	Status
F-2023-0256 - Invalid IBC events handling	Fixed
F-2023-0259 - Malicious Vesting Periods	Fixed
F-2023-0297 - Possibility of duplicate transactions in mempool structure	Fixed
F-2023-0306 - Data race and potential deadlock in PeerState serialization	Fixed

Document

Name	Blockchain Protocol Code Review and Security Analysis Report for Areon
Audited By	Michal Bajor
Approved By	Yaroslav Bratashchuk
Website	https://areon.network/
Changelog	29/12/2023 - Work-in-Progress Draft report
Changelog	19/01/2024 - Preliminary Report

Table to Contents

System Overview

AreonChain is an advanced Layer 1 blockchain protocol, meticulously developed to operate as an independent, EVM (Ethereum Virtual Machine)-compatible solution. At its core, the AreonChain architecture comprises a comprehensive codebase, which encompasses the intricate logic governing the node's functions, its Remote Procedure Call (RPC) API, and an array of related modules. Additionally, it incorporates a suite of essential dependencies, prominently featuring the Cosmos SDK and various other libraries aligned with the Cosmos ecosystem. These dependencies are thoughtfully integrated into the platform as vendored local packages, ensuring a cohesive and stable framework for the AreonChain operations.

The operational excellence of AreonChain is further bolstered by two custom-developed modules, each serving a distinct and critical function within the Areon node:

1. **EVM Module**: This innovative module is expressly designed to support the deployment and interactive engagement with smart contracts tailored for the Ethereum Virtual Machine. By providing a highly compatible and efficient environment, this module enables the flawless execution of Ethereum-derived smart contracts. This compatibility is of paramount importance, especially for developers and users accustomed to the Ethereum platform, as it ensures a smooth transition and operational consistency within the AreonChain ecosystem. The EVM Module is a testament to AreonChain's commitment to interoperability and ease of use, fostering an inclusive and versatile blockchain environment.

2. **FeeMarket Module**: Specializing in the strategic management of transaction fees, this module incorporates the dynamic fee structure outlined in Ethereum Improvement Proposal 1559 (EIP-1559). Its primary function is to adaptively regulate fees, thereby streamlining the transaction process and elevating the network's operational efficiency. By implementing this innovative fee model, AreonChain not only simplifies transaction cost calculations but also ensures a more predictable and equitable fee structure. This approach significantly enhances the user experience, reducing the unpredictability often associated with transaction costs. Moreover, the FeeMarket Module plays a crucial role in maintaining network stability and efficiency, reflecting AreonChain's dedication to providing a user-centric, reliable blockchain infrastructure.

Together, these modules represent the technical prowess and forward-thinking approach of AreonChain. By fusing the familiar benefits of the EVM with the innovative fee management of EIP-1559, AreonChain stands out as a robust, user-friendly blockchain platform. It is an embodiment of the next generation of blockchain technology, designed to meet the evolving needs of users and developers in the ever-expanding blockchain landscape.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's blockchain protocol project. Detailed scoring criteria can be referenced in the corresponding section of the Blockchain Protocol and Security Analysis Methodology.

Documentation quality

The total Documentation Quality score is **9** out of **10**.

The code contains comments and docstrings which document the implementation, however, high-level documentation was missing. It is important to note that a high-level documentation on the architecture itself is present. Given that the AreonChain utilizes a forked blockchain framework, it would be beneficial to have documentation on this, especially highlighting any deviations from the original framework. Additionally, there are no instructions on how to run the node in the README file or where to find this information. During the audit, more detailed technical documentation was provided.

Code quality

The total Code Quality score is **8** out of **10**.

AreonChain effectively utilizes the Go programming language and adheres to its patterns. Furthermore, it follows the coding patterns associated with its underlying blockchain framework, enhancing the readability of the entire codebase. The code contains clear and descriptive comments. However, it is noted that the codebase contains some files (primarily integration tests) that do not compile due to missing dependencies. The code quality score was reduced, as a production-ready software should not contain any code that is not used or doesn't compile.

Architecture quality

The total Architecture Quality score is **10** out of **10**.

The AreonChain architecture is constructed using the Cosmos SDK framework, thereby inheriting its advantageous features. This foundation endows AreonChain with scalability and robust resilience. A key aspect of its design is the integration of the widely recognized and rigorously tested Tendermint consensus protocol, a decision that notably enhances the network's decentralization capabilities. While acknowledging that no system is entirely without vulnerabilities, our comprehensive analysis did not reveal any significant architectural shortcomings in the AreonChain's design.

Security score

Upon auditing, the code was found to contain **0** critical, **0** high, **1** medium, and **3** low severity issues, leading to a security score of **10** out of **10** as all identified issues were correctly fixed.

All identified issues are detailed in the “Findings” section of this report.

General Score

The comprehensive audit of the customer's blockchain protocol yields an overall score of **9.5**. This score reflects the combined evaluation of documentation, code quality, architecture quality, and security aspects of the project.

Findings

Vulnerability Details

F-2023-0259 - Malicious Vesting Periods - Medium

Description:

AreonChain allows for creating Vesting Periods. Those Vesting Periods are validated in the `ValidateBasic` function, in the `sdk/x/auth/vesting/types/msgs.go`. The following function is responsible for this validation:

```
func (msg MsgCreatePeriodicVestingAccount) ValidateBasic() error {
    from, err := sdk.AccAddressFromBech32(msg.FromAddress)
    if err != nil {
        return err
    }
    to, err := sdk.AccAddressFromBech32(msg.ToAddress)
    if err != nil {
        return err
    }
    if err := sdk.VerifyAddressFormat(from); err != nil {
        return sdkerrors.Wrapf(sdkerrors.ErrInvalidAddress, "invalid sender addre
    }

    if err := sdk.VerifyAddressFormat(to); err != nil {
        return sdkerrors.Wrapf(sdkerrors.ErrInvalidAddress, "invalid recipient ad
    }

    if msg.StartTime < 1 {
        return fmt.Errorf("invalid start time of %d, length must be greater than
    }

    for i, period := range msg.VestingPeriods {
        if period.Length < 1 {
            return fmt.Errorf("invalid period length of %d in period %d, length m
        }
    }

    return nil
}
```

However, it was observed that the validation is not implemented in a sufficient way. Namely, the function verifies the validity of source and destination addresses along with Vesting Period's length. Some of the necessary checks are missing, which results in a possibility of creating a

Vesting Period that would allow deposits but not allow withdrawals. As a consequence, should a user deposit funds into an account with malicious Vesting Period, those funds are locked forever.

Status: Fixed

Classification

Severity: Medium

Impact: 5/5

Likelihood: 1/5

Recommendations

Recommendation: It is recommended to implement additional checks on the Vesting Periods. An exemplary fix, would be to make the `for` loop in the `ValidateBasic` function look like this:

```
for i, period := range msg.VestingPeriods {
    if period.Length < 1 {
        return fmt.Errorf("invalid period length of %d in period %d, length r
    }

    if !period.Amount.IsValid() {
        return sdkerrors.ErrInvalidCoins.Wrap(period.Amount.String())
    }

    if !period.Amount.IsAllPositive() {
        return sdkerrors.ErrInvalidCoins.Wrap(period.Amount.String())
    }
}
```

F-2023-0256 - Invalid IBC events handling - Low

Description:

The IBC `RecvPacket` function defines an RPC handler for receiving IBC packets. The function executes a callback and commits state changes only if the acknowledgement is successful. However, if the acknowledgement is not successful it still emits all of the events. The following code snipper is responsible for the described logic:

```
// Perform application logic callback
//
// Cache context so that we may discard state changes from callback if the ac
cacheCtx, writeFn = ctx.CacheContext()
ack := cbs.OnRecvPacket(cacheCtx, msg.Packet, relayer)
if ack == nil || ack.Success() {
    // write application state changes for asynchronous and successful acknow
    writeFn()
} else {
    // NOTE: The context returned by CacheContext() refers to a new EventMana
    // Events should still be emitted from failed acks and asynchronous acks
    ctx.EventManager().EmitEvents(cacheCtx.EventManager().Events())
}
```

The issue revolves around inconsistencies between the state and logs (i.e. events). Every component that is not checking the chain state and uses events as the source of information might consider that certain actions took place, while they did not.

Status: Fixed

Classification

Severity: Low

Impact: 1/5

Likelihood: 2/5

Recommendations

Recommendation: It is recommended to remove the `else` statement starting in the `libs/ibc-go/v6/modules/core/keeper/msg_server.go:414` line.

F-2023-0297 - Possibility of duplicate transactions in mempool structure - Low

Description:

The `mempool` uses two data structures to maintain information about pending transactions - a map and a list. A map contains an index of a transactions in a list. Both of those structures are meant to be in-sync. However, it is possible that they become out-of-sync which leads to a scenario where there are multiple copies of a single transaction in the mempool. The map tracks only a single index, hence it is not possible to remove all copies of a given transaction from the list (even when one of the duplicates was already committed to a block).

The `resCbFirstTime` function is a callback that is called after the node received and checked the transaction for the first time. The current implementation is as follows:

```
func (mem *CListMempool) resCbFirstTime(
    tx []byte,
    peerID uint16,
    peerP2PID p2p.ID,
    res *abci.Response,
) {
    switch r := res.Value.(type) {
    case *abci.Response_CheckTx:
        var postCheckErr error
        if mem.postCheck != nil {
            postCheckErr = mem.postCheck(tx, r.CheckTx)
        }
        if (r.CheckTx.Code == abci.CodeTypeOK) && postCheckErr == nil {
            // Check mempool isn't full again to reduce the chance of exceeding
            // limits.
            if err := mem.isFull(len(tx)); err != nil {
                // remove from cache (mempool might have a space later)
                mem.cache.Remove(tx)
                mem.logger.Error(err.Error())
                return
            }

            memTx := &mempoolTx{
                height:    mem.height,
                gasWanted: r.CheckTx.GasWanted,
                tx:        tx,
            }
            memTx.senders.Store(peerID, true)
            mem.addTx(memTx)
            mem.logger.Debug(
                "added good transaction",
                "tx", types.Tx(tx).Hash(),
                "res", r,
                "height", memTx.height,
                "total", mem.Size(),
            )
            mem.notifyTxsAvailable()
        } else {
            // ignore bad transaction
            mem.logger.Debug(
                "rejected bad transaction",
                "tx", types.Tx(tx).Hash(),
                "peerID", peerP2PID,
                "res", r,
                "err", postCheckErr,
            )
            mem.metrics.FailedTxs.Add(1)

            if !mem.config.KeepInvalidTxsInCache {
                // remove from cache (it might be good later)
                mem.cache.Remove(tx)
            }
        }

    default:
        // ignore other messages
    }
}
```

The callback makes sure that the mempool is not full, however it does not implement any other checks.

If that scenario occurs, the only way to remove the duplicate transactions is to restart the node. If the node is not restarted, it will keep on gossiping the transaction to its peers indefinitely. An attacker might also try to cause a Denial-of-Service condition on the target node by deliberately creating duplicate transactions. This scenario happens when the node's cache overflows. The cache can be increased to make it more challenging for an intentional exploit attempt to be successful.

Status: Fixed

Classification

Severity: Low

Impact: 3/5

Likelihood: 1/5

Recommendations

Recommendation: It is recommended to add a code responsible for manually making sure that there are no duplicate transactions present in the mempool. An exemplary code snippet implementing such a mechanism is as follows:

```
if e, ok := mem.txsMap.Load(types.Tx(tx).Key()); ok {
    memTx := e.(*clist.CElement).Value.(*mempoolTx)
    memTx.addSender(txInfo.SenderID)
    mem.logger.Debug(
    "transaction already there, not adding it again",
    "tx", types.Tx(tx).Hash(),
    "res", r,
    "height", mem.height,
    "total", mem.Size(),
    )
    return
}
```

that should be placed in the `resCbFirstTime` function just after checking if mempool is full.

F-2023-0306 - Data race and potential deadlock in PeerState serialization - Low

Description: It was observed that the `PeerState` structure does not implement the `MarshalJSON` method, but the `ToJSON` method instead. The `ToJSON` implementation is as follows:

```
func (ps *PeerState) ToJSON() ([]byte, error) {
    ps.mtx.Lock()
    defer ps.mtx.Unlock()

    return cmtjson.Marshal(ps)
}
```

However, because no `MarshalJSON` method is present, the JSON encoder uses a reflection mechanism to encode the `PeerState` value. Reflection does not acquire the lock, which results in a data race. Such a data race happens when logger executes an unsynchronised read while concurrent and locked writes are performed (for instance via the `SetHasProposal` and `SetHasVote` functions). It is important to note that this issues is present if logging level is set to `Debug`.

Status: Fixed

Classification

Severity: Low

Impact: 3/5

Likelihood: 1/5

Recommendations

Recommendation: It is recommended to implement a fix consisting of two distinct changes:

1. Rename the `ToJSON` method to `MarshalJSON`, so that reflection mechanism is not used anymore.
2. Make sure that there are no undesirable recursive calls during serialization process.

An exemplary code fixing this issue and adhering to the rules described above is as follows:

```
func (ps *PeerState) MarshalJSON() ([]byte, error) {
    ps.mtx.Lock()
    defer ps.mtx.Unlock()


    type jsonPeerState PeerState
    return cmtjson.Marshal((*jsonPeerState)(ps))
}
```

Observation Details

F-2024-0444 - Simulation tests do not use address prefix indented for the chain - Info

Description: It was observed that the simulation tests define an account keeper like so:

```
app.AccountKeeper = authkeeper.NewAccountKeeper(
    appCodec, keys[authtypes.StoreKey], app.GetSubspace(authtypes.ModuleName),
    authtypes.ProtoBaseAccount, maccPerms, sdk.Bech32MainPrefix,
)
```

It uses the `sdk.Bech32MainPrefix` which is set to `cosmos`.

However, the real application is defining the account keeper in the following manner:

```
app.AccountKeeper = authkeeper.NewAccountKeeper(
    appCodec, keys[authtypes.StoreKey],
    app.GetSubspace(authtypes.ModuleName),
    ethermint.ProtoAccount,
    maccPerms,
    sdk.GetConfig().GetBech32AccountAddrPrefix(),
)
```

It is using the `sdk.GetConfig().GetBech32AccountAddrPrefix()`, which in turn is set to `areon`. This is not a security issue, however it points at lack of consistency across the codebase.

Status: Fixed

Recommendations

Recommendation: It is recommended to keep the address prefix consistent across the codebase. It is a good practice to make tests mimic production environment as closely as possible.

Appendix 1. Severity Definitions

Severity	Description
Critical	Vulnerabilities that can lead to a complete breakdown of the blockchain network's security, privacy, integrity, or availability fall under this category. They can disrupt the consensus mechanism, enabling a malicious entity to take control of the majority of nodes or facilitate 51% attacks. In addition, issues that could lead to widespread crashing of nodes, leading to a complete breakdown or significant halt of the network, are also considered critical along with issues that can lead to a massive theft of assets. Immediate attention and mitigation are required.
High	High severity vulnerabilities are those that do not immediately risk the complete security or integrity of the network but can cause substantial harm. These are issues that could cause the crashing of several nodes, leading to temporary disruption of the network, or could manipulate the consensus mechanism to a certain extent, but not enough to execute a 51% attack. Partial breaches of privacy, unauthorized but limited access to sensitive information, and affecting the reliable execution of smart contracts also fall under this category.
Medium	Medium severity vulnerabilities could negatively affect the blockchain protocol but are usually not capable of causing catastrophic damage. These could include vulnerabilities that allow minor breaches of user privacy, can slow down transaction processing, or can lead to relatively small financial losses. It may be possible to exploit these vulnerabilities under specific circumstances, or they may require a high level of access to exploit effectively.
Low	Low severity vulnerabilities are minor flaws in the blockchain protocol that might not have a direct impact on security but could cause minor inefficiencies in transaction processing or slight delays in block propagation. They might include vulnerabilities that allow attackers to cause nuisance-level disruptions or are only exploitable under extremely rare and specific conditions. These vulnerabilities should be corrected but do not represent an immediate threat to the system.

Appendix 2. Scope

The scope of the project includes the following components from the provided repository:

Scope Details	
Repository	https://github.com/Areon-Network/AreonChain
Commit	acf467ae19bcc740a9a28b51795e
Whitepaper	https://areon.network/docs/areon-whitepaper.pdf
Requirements	
Technical Requirements	

Components in Scope

SDK and cryptography

- Analysis of changes introduced since SDK fork
- Analysis of security fixes in the later versions of Cosmos SDK

Libraries

- Analysis of copied libraries and changes introduced since fork
- Analysis of security fixes in the later versions of copied libraries

Custom modules

- Review of evm and feemarket modules
- Review of app initialisation and configuration

RPC

- Review of RPC API