// Cosmos Security Assessment

09.16.2024 - 10.18.2024

mezod

Mezo

HALBORN

mezod - Mezo

Prepared by: HALBORN

Last Updated 11/19/2024

Date of Engagement by: September 16th, 2024 - October 18th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	0	2	1	0	0

TABLE OF CONTENTS

1. Introduction

Mezo engaged Halborn to conduct a security assessment on their Cosmos project, beginning on September 16, 2024, and ending on October 28, 2024. The security assessment was scoped to cover their `mezod` GitHub repository, located at https://github.com/mezo-org/mezod/tree/main with commit ID `527a9d5a9e7a4823fa21b6d4c41b36b2840ef6e3`.

2. Assessment Summary

The team at Halborn was provided five weeks for the engagement and assigned one full-time security engineer to assess the security of the Cosmos project. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that the **Golang components** operate as intended.
- Identify potential security issues.
- Identify lack of best practices within the codebase.
- Identify systematic risks that may pose a threat in future releases.

In summary, Halborn identified some security issues that were mostly addressed by the `Mezo team`.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the custom modules. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of structures and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment :

- Research into architecture and purpose.
- Static Analysis of security for scoped repository, and imported functions. (e.g., `staticcheck`, `gosec`...)
- Manual Assessment for discovering security vulnerabilities on the codebase.
- Ensuring the correctness of the codebase.
- Dynamic Analysis of files and modules in scope.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: mezod

(b) Assessed Commit ID: 527a9d5

(c) Items in scope:

- precompile/validatorpool/application.go
- precompile/validatorpool/IValidatorPool.sol
- precompile/validatorpool/owner.go
- precompile/validatorpool/validator.go
- precompile/validatorpool/validatorpool.go
- x/poa/keeper/abci.go
- x/poa/keeper/application.go
- x/poa/keeper/genesis.go
- x/poa/keeper/historical_info.go
- x/poa/keeper/keeper.go
- x/poa/keeper/owner.go
- x/poa/keeper/params.go
- x/poa/keeper/query_server.go
- x/poa/keeper/validator.go

Out-of-Scope:

REMEDIATION COMMIT ID:

- https://github.com/mezo-org/mezod/pull/330/commits/76cce191a89420290bd9f839b75c219600f44a9b
- https://github.com/mezo-org/mezod/pull/333/commits/e1fc9057ec49c598a173fb8ce0fe1794930e4e35

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	2	1	0	0

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
RESOURCE EXHAUSTION BY ABUSING APPLICATION'S DESCRIPTIONS	HIGH	SOLVED - 11/14/2024
RESOURCE EXHAUSTION BY SPAMMING APPLICATIONS	HIGH	PARTIALLY SOLVED - 11/18/2024
ALL FUNCTIONS DEFAULT TO THE SAME GAS COSTS	MEDIUM	RISK ACCEPTED

7. FINDINGS & TECH DETAILS

7.1 RESOURCE EXHAUSTION BY ABUSING APPLICATION'S DESCRIPTIONS

// HIGH

Description

As there is no limit on how long the description of an application is, plus the fact that the amount of gas required by a method in its default implementation does not account for the amount of bytes brought from storage to memory, it is possible to send an application with an extremely large description (paying for it, yes) and after that, spam the low-cost function `application` to bring the description back and force from the database, without paying for it, making the node exhaust the memory assigned to its process **OR** suffer from a downtime due to the lower access times to the database to bring such a big amount of data again and again.

Proof of Concept

The function responsible of submitting an application is `submitApplication`, defined as follows:

```
func (m *SubmitApplicationMethod) Run(context *precompile.RunContext, inputs precompile.MethodInputs) (precompi
    // check method inputs
    if err := precompile.ValidateMethodInputsCount(inputs, 2); err != nil {
        return nil, err
    }

    consPubKeyBytes, ok := inputs[0].([32]byte)
    if !ok {
        return nil, fmt.Errorf("consPubKey argument must be type bytes32")
    }

    description, ok := inputs[1].(Description)
    if !ok {
        return nil, fmt.Errorf("description argument must be type Description")
    }

    operator := context.MsgSender()

    // Here we assume consPubKeyBytes is a valid ED25519 key, without performing any additional validation.
    // We may need to add validation here. @ missing validation, una signature que no es la suya ??
    tmpk := ed25519.PubKey(consPubKeyBytes[:])
    consPubKey, err := cryptocdc.FromCmtPubKeyInterface(tmpk)
    if err != nil {
        return nil, err
    }

    validator, err := poatypes.NewValidator(
        types.ValAddress(precompile.TypesConverter.Address.ToSDK(operator)),
        consPubKey,
        poatypes.Description(description),
    )
    if err != nil {
        return nil, err
    }

    err = m.keeper.SubmitApplication(
        context.SdkCtx(),
        precompile.TypesConverter.Address.ToSDK(context.MsgSender()),
        validator,
    )
    if err != nil {
        return nil, err
    }

    // emit event
    err = context.EventEmitter().Emit(
        NewApplicationSubmittedEvent(
            operator,
            consPubKeyBytes,
            description,
        ),
    )
    if err != nil {
        return nil, fmt.Errorf("failed to emit ApplicationSubmitted event: [%w]", err)
    }

    return precompile.MethodOutputs{true}, nil
}
```

it can be seen that, neither in the code nor in the called functions, it checks the length of the `Description` field against an upper limit. If we add the fact that the function `application` does not charge for the ***"used memory",*** it is possible to call it again and again to bring the description from the database (I/O operation, pretty slow in computing time terms) and bring the node to a downtime, possibly crashing it.

BVSS

AO:A/AC:M/AX:L/C:N/I:N/A:C/D:N/Y:N/R:N/S:C (8.3)

Recommendation

Put a limit to the length of the application's description and charge in the gas costs of the function the amount of memory required to account for runtime variables, such as the description of the given application.

Remediation

SOLVED: The issue was solved by checking if the size of the given `description`'s fields are larger than `MaxValidatorDescriptionLength`, returning an error if such a condition is true.

Remediation Hash

https://github.com/mezo-org/mezod/pull/330/commits/76cce191a89420290bd9f839b75c219600f44a9b

7.2 RESOURCE EXHAUSTION BY SPAMMING APPLICATIONS

// HIGH

Description

Due to the flawed gas estimation of the precompile functions, which do not account for the real costs of calling a given method, it is possible to bring the node to a downtime by spamming the network with *"null"* applications (from different accounts due to the checks in place) and calling the `applications` method again and again to make the node loop through all the invalid applications without being charged for that *"looping"* overhead.

Proof of Concept

As there is no limit in the amount of applications (only the amount of validators) plus there is no way to remove old ones or invalid ones from the mapping, then there will be a time where the array of applications (either from normal use or malicious use) will be so large that the node will spend way too much time looping through them. As the required gas is the default one, which does not account for loop iterations, then users do not pay for them, which makes all the parts for a DOS of the node as described in the previous section (possibly bringing the node to a halt due to exhausting the resources assigned by the OS to the node process).

BVSS

AO:A/AC:M/AX:L/C:N/I:N/A:C/D:N/Y:N/R:N/S:C (8.3)

Recommendation

There are many fixes to this, but we recommend the following:

- Implement a privileged method to remove applications
- Change the default required gas implementation to account for the executed loops
- Implement the fixes recommended in the issue *"All functions default to the same gas costs"*

Remediation

PARTIALLY SOLVED: The issue was solved as recommended, with additional changes in https://github.com/mezo-org/mezod/pull/334, which implements the recommendation:

- Implement a privileged method to remove applications

Remediation Hash

https://github.com/mezo-org/mezod/pull/333/commits/e1fc9057ec49c598a173fb8ce0fe1794930e4e35

7.3 ALL FUNCTIONS DEFAULT TO THE SAME GAS COSTS

// MEDIUM

Description

All precompile's functions have the required gas as the default ones. This is bad as it makes it possible for malicious actors to spam costly transactions (in terms of computer power) whilst not paying them fully, which brings down throughput and reduces performance, leading to downtimes in transactions processing.

Proof of Concept

The required gas for all functions default to the following code:

```go
func (m *ValidatorMethod) RequiredGas(_ []byte) (uint64, bool) {
    // Fallback to the default gas calculation.
    return 0, false
}
```

which makes the `Contract` implementation call `DefaultRequiredGas` for the given method:

```go
requiredGas, ok := method.RequiredGas(methodInputArgs)
    if !ok {
        // Fall back to default required gas if method does not determine
        // the required gas by itself.
        requiredGas = DefaultRequiredGas(
            store.KVGasConfig(),
            method.MethodType(),
            methodInputArgs,
        )
    }
```

which is implemented as follows:

```go
func DefaultRequiredGas(
    gasConfig store.GasConfig,
    methodType MethodType,
    methodInputArgs []byte,
) uint64 {
    methodInputArgsByteLength := uint64(len(methodInputArgs))

    costFlat := store.Gas(0)
    costPerByte := store.Gas(0)

    switch methodType {
    case Read:
        costFlat, costPerByte = gasConfig.ReadCostFlat, gasConfig.ReadCostPerByte
    case Write:
        costFlat, costPerByte = gasConfig.WriteCostFlat, gasConfig.WriteCostPerByte
    }

    return costFlat + (costPerByte * methodInputArgsByteLength)
}
```

which does not account for cold accesses, nor loop iterations nor memory needed for the correct allocation of runtime variables, although it is expected in the Cosmos implementation of the struct `GasConfig`:
https://github.com/cosmos/cosmos-sdk/blob/22231f7c6d23393284fcd2ac00e6746458e6b738/store/types/gas.go#L221

```go
// GasConfig defines gas cost for each operation on KVStores
type GasConfig struct {
    HasCost          Gas
    DeleteCost       Gas
    ReadCostFlat     Gas
    ReadCostPerByte  Gas
    WriteCostFlat    Gas
    WriteCostPerByte Gas
    IterNextCostFlat Gas
}
```

That makes it possible for attackers to make the nodes of the network process heavy workloads of work (as in the other issues of this report, which due to this issue makes them cheaper) whilst not paying for the execution cost fully.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:C (6.3)

Recommendation

Implement the correct gas cost of each function based on the number of storage reads and writes, number of iterations of runtime loops as well as memory expansion needed in dynamic data structures.

Remediation

RISK ACCEPTED: The issue was accepted as this functionality is intended in the current release. From the project's words:

> All write functions but `submitApplication` are callable by the PoA owner or existing validators so their cost should be predictable and small. Moreover, given the temporary nature of the PoA module, we decided to use a simple gas model and avoid unnecessary complexity here.

8. AUTOMATED TESTING

Overall, the reported issues were not mostly low/informational issues that did not pose a real threat to the system, so they were not considered to be part of this report.
