Audit Report

Evmos EVM Extensions

v1.0

July 8, 2023

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Tharsis Labs Ltd. to perform a security audit of the Evmos precompiles.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following targets:

Repository	https://github.com/evmos/precompiles
Commit	`bb9002a5c4f52c0eb509ef4e46d32ae480662836`
Scope	All code was in scope.

Repository	https://github.com/evmos/evmos
Commit	`118f5097c6ea8acfd26a9ad236d6a1d0fd27fd7a`
Scope	The changes compared to commit `45cd70ab6f50d04a5ddb60e3043fb83657c1c15a` were in scope.

Repository	https://github.com/evmos/go-ethereum
Commit	`a4867b20faf524eeb4d35a459a10e7144285c915`
Scope	The changes compared to commit `452a12aa7903209713ccdc54af65a1a31e73190d` were in scope.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Areas of Special Interest

During the line-by-line analysis, the auditor team has paid special attention to the following areas:

Gas Consumption

The Evmos precompiles implement the `RequiredGas` function to inform go-ethereum of the required amount of gas to consume. This adheres to the go-ethereum convention also utilized for its built-in precompiles. `RequiredGas` evaluates the required execution gas based on the default gas costs from the Cosmos SDK `KVStore`, which considers input complexity.

Additionally, gas is charged during the `Run` function execution depending on the precompile logic and its interactions with Cosmos SDK modules. Out-of-gas errors are caught through `defer` instructions and reflected in the go-ethereum `Contract` struct.

Similarly, at the end of the execution, the gas consumed in the `GasMeter` is reflected in the go-ethereum `Contract` struct.

Events

Evmos precompiles emit events when invoked by adding a `Log` item to the state. The `Log` data structure consists of an `Address` field containing the address of the precompile that generated the event, `Topics` containing the indexed event parameters represented as 32-byte hashes, `Data` containing the ABI encoded event data, and `BlockNumber` containing the block number of the block in which the event was generated. Event parameters are converted to topics by using the `MakeTopic` function in `common/abi.go`, taken from the go-ethereum codebase.

State Management

Evmos maintains a list of journal logs to track state modifications, which is adapted from the go-ethereum journal implementation. Revisions are used to revert the state to a specific point in time, identified by a unique ID and journal index. If the EVM encounters an error, the state reverts to the snapshot captured at the beginning of the transaction. If the transaction is successful, the dirty states are committed to the keeper. As the EVM and Cosmos SDK maintain separate states during execution, inconsistencies between the two states may occur and need to be handled accordingly.

Common EVM precompile vulnerabilities

Precompiles of other EVM implementations have been affected by multiple severe exploits in the past:

1. **Moonbeam**
   The precompile contract implementing an ERC-20 interface did not account for `DELEGATECALL`, allowing an attacker with a malicious contract to impersonate its caller and steal funds.
2. **Avalanche**
   The `NativeAssetCall` precompile allows calling the provided `to` address with the `callData` parameter while forwarding the caller address of the precompile. This allows an attacker to impersonate the caller, potentially stealing funds.
3. **Aurora**
   By using `DELEGATECALL`, the `ExitToNear` precompile can be tricked into thinking Ether was sent to it, while in reality, Ether was not passed to the precompile. Resulting in a successful withdrawal to NEAR, while retaining the Ether on Aurora, effectively doubling the attacker's original balance.

To protect against these types of attacks, Evmos precompiles' `Run` method accepts a `readOnly` boolean argument which prevents state transitions (i.e., state writes) when executed via `DELEGATECALL`, `STATICCALL`, or `CALLCODE`. Hence, a precompile method that leads to a state transition, determined by the precompiles' `IsTransaction` method, can only be executed via `CALL` (i.e., when the `readOnly` argument is `false`). Otherwise, the precompile will return an error.

We have reviewed all precompiles within the scope of this audit for the above-mentioned exploits and vulnerabilities related to `DELEGATECALL`, `STATICCALL`, and `CALLCODE`. Based on the resources allocated for this audit, we did not identify any issues.

Functionality Overview

EVM extensions allow for the creation of custom precompiled smart contracts in the Evmos EVM. These extensions, unlike built-in precompiles in the go-ethereum EVM, have the capacity to read and modify state, as well as access Cosmos SDK functionality from within the EVM.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium	The Evmos precompiles involve code spread across multiple repositories (precompiles, evmos, go-ethereum). This poses a risk of inadvertently introducing errors while updating one of the repositories, which could break compatibility with code in another repository. Although the existing test suite is sufficient, we recommend enhancing test coverage and implementing extensive integration tests to minimize regression risks. Diverging behavior of the forked and the upstream go-ethereum or Cosmos SDK repositories should be kept at a minimum and thoroughly documented.
Code readability and clarity	Medium-High	Overall, the code is of high quality. However, we suggest evaluating the remaining TODO comments and addressing them accordingly. Rather than relying heavily on code comments, which may become outdated with subsequent code modifications, we recommend using self-descriptive function and variable names.
Level of documentation	High	-
Test coverage	Medium-High	`make test-unit-cover` reports an average test coverage for the `precompiles` repository of 72%.

Summary of Findings

No	Description	Severity	Status
1	Balance updates performed from precompiles are overwritten if the precompile is executed in a transaction with other state change logic	Critical	Resolved
2	User distribution authorizations can be misused by malicious contracts	Minor	Resolved
3	The bech32 precompiler is not loaded in the EVM	Minor	Acknowledged
4	`Validators` and `Redelegations` queries are not callable from evm precompiles	Minor	Resolved
5	Queries do not support pagination, enabling attackers to DOS the chain	Minor	Resolved
6	The `Approve` function does not allow fine-grained `Coin` allowances, allowing more `Coins` than intended in case of multiple messages	Minor	Acknowledged
7	Inconsistency in delegation shares amount in EVM event compared to Cosmos SDK's event	Minor	Resolved
8	Unused `Precompile.IsStateful` function in precompile contracts	Minor	Resolved
9	Inconsistent read-only behavior with precompiles using `CALLCODE`	Minor	Resolved
10	Required gas for precompiles grows rapidly with increasing input length	Minor	Resolved
11	Missing prefix value check can lead to misleading results	Informational	Acknowledged
12	The bech32 precompiles' `baseGas` is not validated to be strictly positive	Informational	Resolved
13	Staking and distribution precompile ABIs are hardcoded, which goes against best practices	Informational	Resolved
14	Outdated comments	Informational	Resolved

Detailed Findings

1. Balance updates performed from precompiles are overwritten if the precompile is executed in a transaction with other state change logic

Severity: Critical

The `Delegate` and `WithdrawDelegatorRewards` precompiles defined in `precompiles/staking/tx.go:59` and `precompiles/distribution/tx.go:67` are not correctly committing balance changes if the caller function performs state changes.

Consequently, attackers could craft a contract where it is possible to delegate to a validator without having the delegated amount deducted from their balance.

Additionally, users that interact with a contract leveraging the `WithdrawDelegatorRewards` precompile could lose their rewards, depending on the contract implementation.

A test case reproducing the issue is provided in Appendix 1.

This issue as well as the test case in Appendix 1 has been disclosed by the Evmos team to Oak Security during the audit.

Recommendation

We recommend committing the balance change in the `stateDB` after the execution of the precompiles.

We also recommend documenting this state update behavior in order to prevent this issue from re-appearing with new precompiles.

Status: Resolved

2. User distribution authorizations can be misused by malicious contracts

Severity: Minor

The state transition functions of the `distribution` module require the user to authorize the caller to execute the `SetWithdrawAddress`, `WithdrawDelegatorRewards`, and `WithdrawValidatorCommision` functions. Failure to do so would result in the transaction's inability to interact with the distribution precompile.

However, the distribution precompile lacks the functionality to revoke authorization once granted, resulting in smart contracts having access granted until the authorizations' expiry.

Although a user can revoke the authorization directly through the Cosmos SDK's authz module, this defeats the purpose of having the precompile in the first place.

Recommendation

We recommend adding revoke authorization functionality to the distribution precompile.

Status: Resolved

3. The bech32 precompiler is not loaded in the EVM

Severity: Minor

The `AvailablePrecompiles` function, defined in `x/evm/keeper/precompiles.go:20-41`, initializes the list of available precompiles.

However, the bech32 precompile is not added to the list, which renders the precompile inaccessible in the EVM.

Additionally, in `precompiles/bech32/bech32.go:63`, the `Address` function does not return a valid precompile address.

Consequently, the bech32 precompile cannot be executed.

Recommendation

We recommend defining a valid address for the bech32 precompile and initializing it in the `AvailablePrecompiles` function.

Status: Acknowledged

The client will add the bech32 precompile at a later point. So far, it has been used as an example for stateless precompiles.

4. `Validators` and `Redelegations` queries are not callable from EVM precompiles

Severity: Minor

The `validators` and `redelegations` functions are available in the staking precompile at `precompiles/staking/Staking.sol:201` and `227`. However, the implementation of both functions in `precompiles/staking/tx.go` is missing, rendering the functions uncallable from EVM-based smart contracts.

Recommendation

We recommend implementing the `validators` and `redelegations` functions in `precompiles/staking/tx.go` to enable access to the corresponding staking query functions.

Status: Resolved

5. Queries do not support pagination, enabling attackers to DOS the chain

Severity: Minor

The query implementations do not support result pagination. This could be problematic since some of the implemented Cosmos SDK queries could return a large number of items. Some examples are the `ValidatorSlashes` and `DelegationRewards` queries defined in `precompiles/distribution/distribution.go:160`.

Additionally, gas is not charged depending on the size of the query result size, but rather on the query input size.

This could allow malicious actors to execute computationally and memory-heavy queries with a disproportionate gas cost, which could slow down block production up to the point where the chain halts.

Recommendation

We recommend implementing pagination functionality and reasonable limits for queries.

Status: Resolved

6. The `Approve` function does not allow fine-grained `Coin` allowances, allowing more `Coins` than intended in case of multiple messages

Severity: Minor

The `Approve` function defined in `precompiles/staking/approve.go:32` iterates through a list of user-provided Cosmos SDK message types to allow a grantee to execute them on behalf of the user with the amount of user-specified `Coins`.

However, because of the chosen input data structure, an encoded `(Address, Coin, []string)` tuple, it is not possible to specify how many `Coins` are to be used for each individual message.

This results in allowing each message type with the `Coin` defined in the input parameter.

Consequently, the total allowance will be different from the specified input since it will be equal to the cardinality of messages multiplied by the provided `Coin`.

For example, a message with an allowance of `1ucoin` and four message types will result in a total allowance of `4ucoin`.

Recommendation

We recommend allowing the user to specify the `Coins` allowance for each provided Cosmos SDK message.

Status: Acknowledged

7. Inconsistency in delegation shares amount in EVM event compared to Cosmos SKD's event

Severity: Minor

The `GetDelegation` function in `precompiles/staking/events.go:132` retrieves the amount of the total shares to emit in the `Delegate` event within the `EmitDelegateEvent` function. However, Cosmos SDK emits the amount of new shares instead of the total amount of shares in `x/staking/keeper/msg_server.go:255`. This inconsistency may cause confusion among users and off-chain components such as indexers processing those EVM and Cosmos SDK events.

Recommendation

We recommend emitting the same value to avoid confusion.

Status: Resolved

8. Unused `Precompile.IsStateful` function in precompile contracts

Severity: Minor

The `IsStateful` function in Evmos's `go-ethereum` fork located in `go-etherum/core/vm/contracts.go:45` defines whether a precompile contract supports state transitions (i.e., stateful) or not (i.e., stateless) and is expected to be implemented by all precompile contracts. However, while the function is implemented in all precompiles, it is not called by the Evmos `go-ethereum` fork as part of the precompile call logic, rendering it ineffective.

Recommendation

We recommend either incorporating the `IsStateful` function and its return value into the precompile call logic or, alternatively, considering removing it.

Status: Resolved

9. Inconsistent read-only behavior with precompiles using `CALLCODE`

Severity: Minor

Evmos precompiles are executed within a forked version of the `go-ethereum` Ethereum execution client by calling the `Precompile.Run` function. This function receives a `readOnly` boolean parameter, which determines if the precompile call is read-only or not. If a precompile is invoked with the `readOnly` parameter set to `true` and the invoked method writes to state, the call will be reverted in `precompiles/staking/staking.go:122`, respectively `precompiles/distribution/distribution.go:104`.

As per the Evmos precompile contract guidelines, precompiles invoked with `DELEGATECALL`, `STATICCALL`, or `CALLCODE` opcodes are expected to exhibit read-only behavior. However, invoking a precompile with the `CALLCODE` opcode does not set the `readOnly` parameter to `true` in the `Precompile.Run` function in `go-ethereum/core/vm/evm.go:275`. Although this discrepancy does not pose an immediate security risk, it deviates from the precompile contract guidelines and could result in unexpected behavior.

Recommendation

We recommend ensuring that precompile contracts invoked with the `CALLCODE` opcode run as read-only calls by setting the `readOnly` parameter to `true`.

Status: Resolved

10. Required gas for precompiles grows rapidly with increasing input length

Severity: Minor

Evmos precompiles calculate the required execution gas in the `RequiredGas` method. The required gas is calculated by multiplying the flat gas cost with the input length, as observed for the staking precompile in `precompiles/staking/staking.go:95` and `98`, as well as for the distribution precompile in `precompiles/distribution/distribution.go:78` and `81`. This calculation differs from that employed in other EVM precompiles, which use a base gas cost along with an additional gas cost per word, as seen in the Evmos `go-ethereum` fork in `core/vm/contracts.go:390`.

Taking the staking precompile as an example, the required gas for a transactional method is calculated in `precompiles/staking/staking.go:95` as follows:

`p.kvGasConfig.WriteCostFlat * uint64(len(argsBz))`

As a result of this calculation, increasing the input length leads to a rapid increase in the gas cost since the flat gas cost is multiplied by the input length.

Recommendation

We recommend re-evaluating the required gas calculation for the staking and distribution precompile and consider incorporating the gas cost per byte into the equation:

`p.kvGasConfig.WriteCostFlat + p.kvGasConfig.WriteCostPerByte * uint64(len(argsBz))`

Status: Resolved

11. Missing prefix value check can lead to misleading results

Severity: Informational

The `HexToBech32` function in `precompiles/bech32/methods.go:24` converts hex addresses to the bech32 format based on a provided prefix. However, the current implementation fails to validate whether the `prefix` value matches any of the underlying chain's address prefixes (i.e., `AccountAddrPrefix`, `ValidatorAddrPrefix`, and `ConsensusAddrPrefix`), potentially causing misleading results.

Recommendation

We recommend validating the provided prefix value to ensure the prefix is either an `AccountAddrPrefix`, `ValidatorAddrPrefix`, or `ConsensusAddrPrefix`.

Status: Acknowledged

The client has assessed that the bech32 precompile is designed to be sufficiently generic, supporting bech32 prefixes beyond "evmos1…", such as "osmo1…", "cosmos1…", etc.

12. The bech32 precompiles' `baseGas` is not validated to be strictly positive

Severity: Informational

The bech32 precompile charges a fixed `baseGas` gas fee for its execution.

However, during its initialization, the `NewPrecompile` function defined in `precompiles/bech32/bech32.go:43`, does not enforce that `baseGas` is a non-zero value.

This enables the EVM initializer to configure the precompile with a zero gas fee allowing potential attackers to leverage it for a DOS attack.

We report this issue as informational since the EVM initialization logic is developed by the same entity as the precompilers.

Recommendation

We recommend validating the `baseGas` parameter to be strictly positive.

Status: Resolved

13. Staking and distribution precompile ABIs are hardcoded, which goes against best practices

Severity: Informational

Precompile contract ABIs are initialized during contract instantiation with the `NewPrecompile` function. The ABIs for the staking and distribution precompiles are currently hardcoded though, instead of being loaded from a JSON file, as seen in `precompiles/staking/staking.go:56` and `precompiles/distribution/distribution.go:40`.

While not a direct security concern, hard coding of generated files goes against best practices, since it can lead to inconsistencies and subtle bugs.

Recommendation

We recommend adopting a consistent approach for ABI initialization by dynamically loading the precompile ABIs from their respective JSON files, following the example set by the bech32 precompile.

Status: Resolved

14. Outdated comments

Severity: Informational

In some parts of the codebase, comments are outdated and may potentially confuse or mislead developers and users. The following incomplete list highlights specific instances of such outdated comments:

- In `precompiles/bech32/bech32.go:61`, the comment inaccurately refers to the staking precompile contract, whereas it should mention the bech32 precompile contract.
- In `precompiles/bech32/bech32.go:67`, the comment incorrectly states that the `IsStateful` function returns `true`. The actual return value is `false`.
- In `precompiles/staking/approve.go:241`, the comment wrongly mentions that the `increaseAllowance` function decreases the allowance amount. In reality, the function increases the allowance amount.

Recommendation

We recommend revising the comments to accurately reflect the current implementation as well as the intended functionality of the code. This will ensure clear and unambiguous communication with developers and users interacting with the codebase.

Status: Resolved

Appendix: Test Cases

1. **Test case for “Balance updates performed from precompiles are overwritten if the precompile is executed in a transaction with other state change logic”**

The following test case reproduces the described issue. While the `testDelegate` function correctly updates the balance, the `testDelegateWithCounter` function does not.

```
import "Staking.sol";

contract Test {

    // A simple counter to be incremented
    uint256 public counter;
    // The Cosmos messages we are authorizing
    string[] private stakingMethods = [MSG_DELEGATE];

    // Approve function that authorizes the staking methods
    function approveRequiredMsgs(uint256 _amount) public {
        bool successStk = STAKING_CONTRACT.approve(msg.sender, _amount,
stakingMethods);
        require(successStk, "Staking Approve failed");
    }

    // A payable function to deposit funds into the smart contract
account
    function deposit() payable public {
    }

    // This version of delegate will correctly create a delegation but
will NOT deduct the balance
    function testDelegateWithCounter(string memory _validatorAddr,
uint256 _amount) public {
        counter += 1;
        approveRequiredMsgs(_amount);
        STAKING_CONTRACT.delegate(address(this), _validatorAddr,
_amount);
    }

    // This version of delegate will correctly create a delegation and
will deduct the balance
    function testDelegate(string memory _validatorAddr, uint256 _amount)
public {
        approveRequiredMsgs(_amount);
        STAKING_CONTRACT.delegate(address(this), _validatorAddr,
_amount);
    }


    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```