code4rena

Nibiru

Smart Contract Security Assessment

Version 2.0

Audit dates: Oct 07 — Oct 17, 2024

Audited by: berndartmueller
jakub-heba

Contents

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About Nibiru

Nibiru Chain is a breakthrough Layer 1 blockchain and smart contract ecosystem providing superior throughput, unparalleled security, and a highperformance EVM execution layer. Nibiru aims to be the most developer-friendly and user-friendly smart contract ecosystem, leading the charge toward mainstream Web3 adoption by innovating at each layer of the

stack: dApp development, scalable blockchain data indexing, consensus optimizations, a comprehensive developer toolkit, and composability across multiple VMs

2.2 Scope

Repository	NibiruChain
Commit Hash	585ebe70900b70ac9aca3add9580a15d283048d5
Mitigation Hash	94db838eb391a0cef6b418af96e27220b3d73bf4

2.3 Audit Timeline

DATE	EVENT
Oct 07, 2024	Audit start
Oct 17, 2024	Audit end
Nov 07, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	4
High Risk	4
Medium Risk	2
Low Risk	6
Informational	3
Total Issues	19

3. Findings Summary

ID	DESCRIPTION	STATUS
C-1	EVM base fee is set to 1 wei instead of 1e12 wei	Resolved

C-2	Remaining EVM gas refund is an order of magnitude higher than expected	Resolved
C-3	Dirty EVM state is not committed to the Cosmos SDK state in the custom precompiles, allowing infinite `unibi` minting	Resolved
C-4	Precompile dirty EVM state applied after Cosmos state changes	Resolved
H-1	On-chain gas limit estimation for internal EVM calls can be exploited to perform DoS attacks	Resolved
H-2	Gas spent in internal EVM calls is not compensated for by the user in `MsgCreateFunToken` and `MsgConvertCoinToEvm` messages	Resolved
H-3	Missing local gas meter in precompiles allows exceeding the precompile call's gas limit	Resolved
H-4	Cosmos SDK state changes are not reverted when EVM state reverts	Resolved
M-1	`RequiredGas()` in funtoken and oracle precomple does not account for gas usage of dynamic types	Resolved
M-2	`Funtoken` precompile unchecked `transfer` return value	Resolved
L-1	Internal EVM call does not update block bloom filter with emitted logs	Resolved
L-2	Less than `1e12` wei NIBI tokens can be transferred from within the EVM which results in lost funds	Acknowledged
L-3	`Oracle` precompile omits `timestamp`	Acknowledged
L-4	Wasm `ExecuteMulti` missing validations	Resolved
L-5	Missing fee validation in `SetParams` keeper	Resolved
L-6	Unsupported weird ERC20 tokens	Acknowledged
I-1	Improve bank coin metadata for ERC-20 funtokens	Resolved
I-2	Multiple functions are not used in the protocol's logic	Acknowledged

I-3	Outdated comment and improper error message	Resolved

4. Findings

4.1 Critical Risk

A total of 4 critical risk findings were identified.

[C-1] EVM base fee is set to 1 wei instead of 1e12 wei

Severity: Critical	Status: Resolved

Context:

- x/evm/keeper/vm_config.go#L30-L36

Description: In `GetEVMConfig()`, the `BaseFee` is set to `evm.BASE_FEE_MICRONIBI = big.NewInt(1)`, which results in the base fee being set to `1 wei` instead of converting the `1unibi` to `1e12 wei`.

This has wide-reaching implications as the base fee is significantly lower than expected.

For example, in `ApplyEvmTx(..)`, converting the transaction via `msg, err := tx.AsMessage(signer, evmConfig.BaseFee)` will incorrectly set `msg.gasPrice` to a lower value than expected, e.g., a gas price of `1 wei` instead of `1e12 wei`.

`go-ethereum@v1.10.27-nibiru/core/types/transaction.go`

```
// If baseFee provided, set gasPrice to effectiveGasPrice.
if baseFee != nil {
  msg.gasPrice = math.BigMin(msg.gasPrice.Add(msg.gasTipCap, baseFee),
msg.gasFeeCap)
}
```

Moreover, `BaseFee` being set to `1` will lead to incorrect calculations in `evmante_can_transfer.go` and many other instances in the codebase.

Recommendation: Consider consistently converting the base fee to `1e12 wei` instead of `1 wei`.

Nibiru: Resolved with PR-2076

Zenith: Verified.

[C-2] Remaining EVM gas refund is an order of magnitude higher than expected

Severity: Critical	Status: Resolved

Context:

- x/evm/keeper/gas_fees.go#L48

Description: `RefundGas()`, which is called in `ApplyEvmTx` after the EVM transaction is executed, refunds the remaining unspent EVM gas via a bank coin transfer.

The refund amount `remaining` is calculated as

```
// Return EVM tokens for remaining gas, exchanged at the original rate.
remaining := new(big.Int).Mul(new(big.Int).SetUint64(leftoverGas),
msg.GasPrice())
```

This amount is denominated in wei units instead of `unibi` due to `msg.GasPrice()` being in wei.

However, when creating the bank coin `refundedCoins`,

```
// positive amount refund
refundedCoins := sdk.Coins{sdk.NewCoin(denom,
sdkmath.NewIntFromBigInt(remaining))}
```

the `remaining` amount is **not** converted to `unibi` (`1unibi` = `1e12 wei` NIBI EVM tokens). As a result, the bank coin transfer will refund the remaining gas in wei units instead of `unibi`, which is an order of magnitude higher.

For example, let's assume the transaction's `gasPrice = 1e12 = 1unibi` (gas price is in wei and has 18 decimals; `1e18` EVM NIBI tokens equal `1e6unibi`) and leftover gas is `50_000`. This results in `remaining = 50_000 * 1e12 = 5e16`.

Consequently, it would refund `5e16 micronibi`, which is equal to `5e10 NIBI` and massively more than expected.

Recommendation: Consider converting the wei amount to native `unibi` units before creating the bank coin to refund the remaining gas.

Nibiru: Addressed here:

- PR-2076
- Info: PR-2059

The pull request switches the function to use the MsgEthereumTx instead of the raw gethcore.Message for the transaction, which allows us to call EffectiveGas* functions like in the Ante Handler. We also labeled the units explicitly to prevent future similar bugs and added tests to verify the behavior.

Zenith: Verified.

[C-3] Dirty EVM state is not committed to the Cosmos SDK state in the custom precompiles, allowing infinite `unibi` minting

Severity: Critical	Status: Resolved

Context:

- x/evm/precompile/wasm.go#L94
- x/evm/precompile/funtoken.go#L70
- x/evm/precompile/oracle.go#L55

Description: The `Run()` function of the custom precompiles retrieve the Cosmos state by `stateDB, ok := evm.StateDB.(*statedb.StateDB)`, which is subsequently used by various Cosmos SDK modules (e.g. bank, wasm).

However, this Cosmos SDK state might be outdated at this point due to any EVM state changes that occurred before the precompile was called. For example, transferring native NIBI tokens via Solidity's `transfer` function would update the EVM's state object for the specific account and deduct the transferred funds. But this balance change is not reflected in the Cosmos SDK account's state.

In essence, the dirty EVM journal entries are not applied to the Cosmos SDK state at the start of the precompile call in `Run()`.

This can be exploited by having a Solidity contract that transfers its native NIBI token balance to another address and then calls the wasm precompile's `IWasm.execute` function with `funds` including the native `unibi` bank coin with a non-zero amount. Internally, the wasm module will operate under the assumption that the caller has sufficient funds and transfers the coin to the called wasm contract. As a result, this results in an infinite mint of the native `unibi` bank coin.

Recommendation: Consider committing the dirty EVM state changes to the Cosmos state, similar to how it is done in Evmos -> precompile.go#L95-L99

Please be aware of not introducing the critical infinite mint vulnerability that was previously found and disclosed by Asymmetric Research (fix).

Overall, it is recommended to closely follow the Evmos precompile implementation to prevent any other potential vulnerabilities as well as tracking future updates to the Evmos precompiles.

Nibiru: Should be resolved by PR-2094 and PR-2095

Zenith: Verified

[C-4] Precompile dirty EVM state applied after Cosmos state changes

Severity: Critical	Status: Resolved

Context:

- funtoken.go#L190

```
// Transfer the bank coin
        err = p.bankKeeper.SendCoinsFromModuleToAccount(ctx,
evm.ModuleName, toAddr, coins)
        if err != nil {
                return nil, fmt.Errorf("send failed for module \"%s\"
(%s): contract caller %s: %w",
                        evm.ModuleName, evm.EVM_MODULE_ADDRESS.Hex(),
caller.Hex(), err,
                )
        }
```

Here, the `bankKeeper` modifies the Cosmos state by sending coins from the module to the user's account without updating the EVM state accordingly.

Description: It was found, that the `bankSend` precompile at `x/evm/precompile/funtoken.go:118` contains an issue due to inconsistent state management between the EVM state and the Cosmos SDK state. The vulnerability arises because balance changes made to the Cosmos state using the `bankKeeper` are not immediately reflected in the EVM state, leading to overwriting of updated balances and state inconsistencies.

When `bankSend` executes `p.bankKeeper.SendCoinsFromModuleToAccount`, it modifies the Cosmos state by transferring coins from the module account to a user account (`toAddr`). However, the EVM state does not immediately recognize this change.

If the Solidity contract invoking this precompile has made prior state changes, for example by modifying account balances or storage variables, these changes are recorded in the EVM's journal. The EVM applies these "dirty" states to the Cosmos state only at the end of the transaction execution.

Because the EVM applies its dirty state after the `bankKeeper` has modified the Cosmos state, any balance changes made by the `bankKeeper` can be overwritten by the outdated EVM state. This leads to discrepancies between the actual account balances and what the EVM perceives.

This vulnerability is similar to the `Precompile dirty EVM state applied after cosmos state changes` issue identified in Evmos (see Evmos Security Advisory GHSA-68fc-7mhg-6f6c). In both cases, the lack of synchronization between the EVM and Cosmos states allows for the EVM's outdated state to overwrite the updated Cosmos state, resulting in exploitable conditions.

It is worth to mention, that other precompiles might be vulnerable to this issue as well. For example, in the `wasm` precompile, when funds are sent to the called wasm contract, they are sent within the Cosmos SDK state, which is out of sync with the EVM state when before the precompile call the native EVM balance is sent via `transfer`.

Recommendation: We recommend, that when performing state transitions, especially those involving account balances, the application should commit the balance change in the `stateDB` after the execution of the precompiles.

Nibiru: Addressed by PR-2095

Zenith: Verifeid

4.2 High Risk

A total of 4 high risk findings were identified.

[H-1] On-chain gas limit estimation for internal EVM calls can be exploited to perform DoS attacks

Severity: High	Status: Resolved

Context:

- x/evm/keeper/erc20.go#L184-L186

Description: `CallContractWithInput()` is called internally when executing messages such as `MsgCreateFunToken`. It estimates the gas limit that will be provided for the actual EVM call. The gas estimation uses a binary search algorithm to find the gas limit that is sufficient for the call to succeed. The maximum gas limit is set to `DefaultEthCallGasLimit uint64 = 25_000_000`.

However, such messages are executed on-chain by all nodes, while the gas estimation requires computational resources, which the user does not pay for. This opens up a DoS vector where a Solidity is purposefully crafted to extract as many computational resources as possible during gas estimation.

I think it's simply not a good idea to estimate gas on-chain. Instead, a hard-coded gas limit should be used that is likely to be sufficient for "honest" erc-20 tokens.

Recommendation: During the execution of messages on-chain, consider using a hard-coded gas limit for internal EVM calls done by `CallContractWithInput()` that is likely to be sufficient for "honest" ERC-20 tokens instead of estimating the gas limit.

Nibiru: Fixed with PR-2089

Zenith: Verified.

[H-2] Gas spent in internal EVM calls is not compensated for by the user in `MsgCreateFunToken` and `MsgConvertCoinToEvm` messages

Severity: High	Status: Resolved

Context:

- x/evm/keeper/msg_server.go#L447
- x/evm/keeper/msg_server.go#L521

Description: Creating a funtoken from an already existing ERC-20 token requires sending a *regular* Cosmos message `MsgCreateFunToken`, which is processed by the `CreateFunToken` keeper function. Moreover, converting an amount of a bank coin that was originally an ERC-20 token back to an ERC-20 token can be done via the `MsgConvertCoinToEvm` message, implemented in the `ConvertCoinToEvm` keeper function.

Internally, an internal EVM call is made by `CallContractWithInput()`, e.g. to retrieve the ERC-20 metadata (name, symbol, and decimals) from the ERC-20 token contract or to transfer the escrowed ERC-20 tokens to the recipient. This internal EVM call uses a `gasLimit` that is by default set to a maximum of 25M. By using binary search, the `gasLimit` is adjusted so that it is sufficiently high to execute the internal EVM call without reverting with an out-of-gas error.

However, contrary to the special `MsgEthereumTx` message that resets the Cosmos SDK gas meter at the end of the EVM execution to the gas spent by the EVM, the `MsgCreateFunToken` message (and also other regular Cosmos SDK messages that internally call the EVM) is not aware of the gas spent by the EVM as the gas meter is not reset to the gas spent in the EVM.

Consequently, the gas spent in the EVM is not compensated for by the user. This can be exploited to DoS the network by deploying a custom ERC-20 token that consumes up to 25M gas in functions like `name()`, `balanceOf(..)` or `transfer(..)` and sending a large number of Cosmos SDK messages that internally call those functions.

Recommendation: Consider resetting the Cosmos SDK gas meter to the gas spent by the EVM (e.g., via `ResetGasMeterAndConsumeGas(..)`) in every message keeper function that internally calls the EVM to prevent DoS attacks by consuming all the gas in the EVM.

Nibiru: Fixed with PR-2089

Zenith: Verified

[H-3] Missing local gas meter in precompiles allows exceeding the precompile call's gas limit

Severity: High	Status: Resolved

Context:

- x/evm/precompile/wasm.go#L97

Description: The wasm precompile retrieves the current context `ctx := stateDB.GetContext()` and passes it on to the wasm module to handle the actual execution. The wasm module internally keeps track of the consumed gas by using the gas meter from the context. However, this gas meter is the overall gas meter of the Cosmos message and not the gas meter that is restricted by the precompile call's gas limit (which is provided by the EVM).

In other words, the precompiles lack creating a "local" gas meter that is initialized to the precompiles gas limit and which can be used to track and limit the consumed gas in the wasm module.

For example, imagine a Solidity contract `A` is called with 1M prepaid gas, which is the contract's gas limit that must not be exceeded. This contract then calls another contract `B.foo()` which internally uses the wasm precompile. `B.foo{ gas: 100_000 }()` is called with an explicit gas limit to prevent the callee (`B`) from using too much gas -> `B.foo()` must stay below 100k gas. Within `B.foo()`, the wasm precompile is called. However, the wasm precompile retrieves the `ctx := stateDB.GetContext()` Cosmos SDK context, which uses the gas meter with the 1M gas limit. So the precompile is able to exceed the 100k gas limit and consume way more than anticipated.

Notably, it cannot consume more than the 1M limit, as it would panic with `ErrorOutOfGas`.

Recommendation: Consider creating a "local" gas meter, similar to the implementation in Evmos, which can be seen in https://github.com/evmos/evmos/blob/0039c64c8201bb0a0c6389712a06ffa50849e8b4/precompiles/common/precompile.go#L96-L104.

Additionally, it is important to apply the spent gas to `contract` via `contract.UseGas(..)` and handle any out-of-gas errors accordingly. See erc20.go#L140-L142 and erc20.go#L149-L153 for details.

Please note that this mechanism is best shared with all precompiles.

Nibiru: Addressed by PR-2093 & PR-2107

Zenith: Verified

[H-4] Cosmos SDK state changes are not reverted when EVM state reverts

Severity: High	Status: Resolved

Context:

- x/evm/precompile/wasm.go#L97
- x/evm/precompile/funtoken.go#L73
- x/evm/precompile/oracle.go#L55

Description: The custom precompiles retrieve the current context by using `ctx := stateDB.GetContext()` in `Run()`. This context is subsequently used and passed to Cosmos SDK functions such as `p.bankKeeper.MintCoins(..)` or `p.Wasm.Execute(..)`, which apply state changes to this context.

However, it is possible that the containing EVM call reverts (e.g. due to a `revert(..)` in the Solidity contract), which will revert the EVM state changes, but misses to revert any Cosmos state changes. This can lead to inconsistencies between the EVM and Cosmos state, which in the worst case and depending on the actual implementation of the affected Solidity contract can be exploited to steal funds.

For example, this issue becomes relevant for Solidity contracts that revert after a call to a Nibiru precompile:

```
contract Test {
  uint256 public counter;

  function foo() {
    try Test(this).bar() {} catch {}
  }

  function bar() {
    counter++;

    // Call precompile that changes Cosmos state

    revert(); // This will revert the `counter` state change, but not the
Cosmos state change done by the precompile
  }
}
```

Recommendation: Instead of re-using the same current context by using `ctx := stateDB.GetContext()`, consider creating a new temporary context to be able to revert the

Cosmos state as well when then EVM state is reverted.

It is recommended to use a similar mechanism as used by Evmos, see precompile.go#L84-L93 and precompile.go#L193-L195, where the journal is adapted to keep track of precompile calls with the snapshot of the Cosmos state, so that it can be reverted when the EVM state is reverted via `RevertToSnapshot()`.

Nibiru: Addressed by PR-2094 & PR-2095

Zenith: Verified.

4.3 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] `RequiredGas()` in funtoken and oracle precomple does not account for gas usage of dynamic types

Severity: Medium	Status: Resolved

Context:

- x/evm/precompile/funtoken.go#L45
- x/evm/precompile/oracle.go#L32

Description: `RequiredGas()` in the precompiles listed above charges a fixed amount of gas. This is problematic as the `bankSend` function accepts a dynamic type parameter, `string memory to`, as the transfer recipient and the oracle's `queryExchangeRate` function accepts the dynamic `string memory pair` parameter.

For example, if the `bankSend` precompile is called with the `to` string set to a **very** large string value, the `DecomposeInput()` call will consume a lot of computational resources to unpack the large `to` value, which the user is not charged for. Similarly, this also applies to the Oracle precompile.

If such a message is executed on chain by all nodes, it will consume gas that was not paid for and this might be used as part of a DoS attack against the network.

Recommendation: If dynamic types are used as function parameters, it is very important to be accurate with gas accounting, as the value can potentially be very large. For example, consider parsing the `input` also in `RequiredGas()` and charge additional gas for dynamic size parameters (e.g. `string`, `bytes`, etc.)

Nibiru: Fixed by PR-2086

- precompile/oracle.go
- precompile/funtoken.go

Zenith: Verified

[M-2] `Funtoken` precompile unchecked `transfer` return value

Severity: Medium	Status: Resolved

Context:

- funtoken.go#L162-L164

```
// Caller transfers ERC20 to the EVM account
        transferTo := evm.EVM_MODULE_ADDRESS
        _, err = p.evmKeeper.ERC20().Transfer(erc20, caller, transferTo,
amount, ctx)
        if err != nil {
                return nil, fmt.Errorf("failed to send from caller to the
EVM account: %w", err)
        }
```

Description: In the `Funtoken` precompile module, the `bankSend` function fails to check the boolean return value of the `ERC20.transfer` method. Specifically, the transfer of ERC20 tokens from the caller to the EVM account is performed without verifying whether the `transfer` was successful:

```
_, err = p.evmKeeper.ERC20().Transfer(erc20, caller, transferTo, amount,
ctx)
if err != nil {
    return nil, fmt.Errorf("failed to send from caller to the EVM
account: %w", err)
}
```

This omission poses a risk to the protocol because, according to the ERC-20 standard, the `transfer` function returns a boolean value indicating the success of the operation, and callers must handle this return value appropriately. Some ERC20 tokens, such as ZRX or BAT, are designed to return `false` in case of an error instead of reverting the transaction. Without checking the returned boolean value, the code may incorrectly assume that the `transfer` was successful even when it was not.

This can lead to inconsistencies in token balances and another security issues. It is worth to mention, that in other parts of the codebase where `transfer` is called, the return value is properly checked to ensure the operation succeeded.

Recommendation: We recommend that the `bankSend` function should be updated to check the boolean return value of the `ERC20.transfer` method validating, if the token transfer was successful. It will handle the case where it returns `false` by aborting the operation and returning an appropriate error message. This can be implemented by modifying the `transfer` call, for example, as follows:

```
success, err := p.evmKeeper.ERC20().Transfer(erc20, caller, transferTo,
amount, ctx)
if err != nil {
    return nil, fmt.Errorf("failed to send from caller to the EVM
account: %w", err)
}
if !success {
    return nil, fmt.Errorf("ERC20 transfer returned false for token
\"%s\"", erc20.Hex())
}
```

Nibiru: Addressed by PR-2090. This solution follows the description of the ERC20 spec and queries the balance before and after the transfer to get the exact amount of tokens received

Zenith: Verified

4.4 Low Risk

A total of 6 low risk findings were identified.

[L-1] Internal EVM call does not update block bloom filter with emitted logs

Severity: Low	Status: Resolved

Context:

- x/evm/keeper/erc20.go#L217-L219

Description: The internal EVM call made by `CallContractWithInput()` does not update the block bloom filter with the emitted EVM logs. Contrary to `ApplyEvmTx` in `x/evm/keeper/msg_server.go#L118-L123`.

Recommendation: To ensure that all EVM logs are consistently included in the block bloom filter, consider updating the block bloom filter with the emitted EVM logs in `CallContractWithInput()`, similar to `ApplyEvmTx`.

Nibiru: Implemented with PR-2089

Zenith: Verified.

[L-2] Less than `1e12` wei NIBI tokens can be transferred from within the EVM which results in lost funds

Severity: Low	Status: Acknowledged

Context:

- x/evm/statedb/state_object.go#L61

Description: Sending less than `1e12 wei` NIBI tokens (1e12 wei NIBI in the EVM equals `1unibi` in Cosmos SDK coin) is prevented by the `ApplyEvmMsg` function within `ParseWeiAsMultipleOfMicronibi()` when supplying the transfer amount in `msg.Value()`:

```
// err if weiInt is too small
tenPow12 := new(big.Int).Exp(big.NewInt(10), big.NewInt(12), nil)
if weiInt.Cmp(tenPow12) < 0 { // @note weiInt < 10^12
        return weiInt, fmt.Errorf(
                "wei amount is too small (%s), cannot transfer less than
1 micronibi. 10^18 wei == 1 NIBI == 10^6 micronibi", weiInt)
}
```

However, it is still possible to send a tiny amount of NIBI tokens (< `1e12`) from within the EVM itself, i.e., via a Solidity `transfer()`. The transfer will succeed, but as soon as the EVM state object is committed to the Cosmos state via `StateDB.Commit()`, it will round down any wei balance that's < 1e12. As a result, the funds are lost.

Nevertheless, as it is just a tiny amount that is potentially lost, we consider this a low-severity issue.

Recommendation: Consider overwriting the `CanTransfer` function in `vm.BlockContext` to prevent sending less than `1e12 wei` NIBI tokens. Or, alternatively, clearly document this behavior to prevent users from losing funds.

Nibiru: This was more an intentional design choice. Acknowledged

[L-3] `Oracle` precompile omits `timestamp`

Severity: Low	Status: Acknowledged

Context:

- oracle.go#L108

```
func (p precompileOracle) queryExchangeRate(
        ctx sdk.Context,
        method *gethabi.Method,
        args []interface{},
        readOnly bool,
) (bz []byte, err error) {
        pair, err := p.decomposeQueryExchangeRateArgs(args)
        if err != nil {
                return nil, err
        }
        assetPair, err := asset.TryNewPair(pair)
        if err != nil {
                return nil, err
        }

        price, err := p.oracleKeeper.GetExchangeRate(ctx, assetPair)
        if err != nil {
                return nil, err
        }

        return method.Outputs.Pack(price.String())
}
```

Description: The `queryExchangeRate` function in the `Oracle` precompile module currently returns only the exchange rate for a given asset pair without including the `timestamp` of when this exchange rate was last updated. This omission poses a risk because it prevents Solidity contracts from verifying the freshness of the exchange rate data they receive.

In financial applications, the timeliness of price data is very crucial - using stale or outdated exchange rates can lead to incorrect calculations and financial losses. By withholding the `timestamp`, the function limits the ability of contracts to perform checks on data validity or to compute time-weighted average prices (TWAP), which require knowledge of when each price point was recorded.

Recommendation: We recommend modyfing the `queryExchangeRate` function to include the `timestamp` of the last update to the exchange rate alongside the `rate` itself. By providing both, Solidity contracts can assess whether the data is sufficiently recent for their purposes and can implement safeguards against using stale prices.

This change will enhance the utility of the `Oracle` precompile by enabling contracts to perform more robust data validation and to calculate TWAPs if needed.

Nibiru: Acknowledged

[L-4] Wasm `ExecuteMulti` missing validations

Severity: Low	Status: Resolved

Context:

- wasm.go#L314

```
wasmExecMsgs, err := p.parseExecuteMultiArgs(args)
        if err != nil {
                err = ErrInvalidArgs(err)
                return
        }
        callerBech32 := eth.EthAddrToNibiruAddr(caller)

        var responses [][]byte
        for _, m := range wasmExecMsgs {
                wasmContract, e :=
sdk.AccAddressFromBech32(m.ContractAddr)
                if e != nil {
                        err = fmt.Errorf("Execute failed: %w", e)
                        return
                }
                var funds sdk.Coins
                for _, fund := range m.Funds {
                        funds = append(funds, sdk.Coin{
                                Denom:  fund.Denom,
                                Amount:
sdk.NewIntFromBigInt(fund.Amount),
                        })
                }
                respBz, e := p.Wasm.Execute(ctx, wasmContract,
callerBech32, m.MsgArgs, funds)
```

Recommendation: The `executeMulti` function in the `wasm` precompile module lacks essential validations that are present in the single-message `execute` function. In the `execute`, validation is performed in a helper it calls - the `parseExecuteArgs` function. This helper includes a call to `msgArgsCopy.ValidateBasic`, which ensures that the message arguments conform to expected formats and constraints before execution.

Specifically, within `parseExecuteArgs`, after extracting the `msgArgs`, the code performs the following validation:

```
msgArgsCopy := wasm.RawContractMessage(msgArgs)
if e := msgArgsCopy.ValidateBasic(); e != nil {
    err = ErrArgTypeValidation(e.Error(), args[argIdx])
    return
}
```

This validation prevents malformed or malicious messages from being processed.

In contrast, the `executeMulti` function calls a `parseExecuteMultiArgs` function, which does not include similar validation. The `parseExecuteMultiArgs` parses the arguments but does not perform a `ValidateBasic` call on the message arguments. Consequently, invalid or improperly formatted messages could be passed to the `wasm.Execute` function without prior checks.

Recommendation: We recommend implementing thorough validation of message arguments within the `executeMulti` function, similar to the validation logic present in the `parseExecuteArgs` function called by `execute`. This can be achieved by implementing a `ValidateBasic` call for each message in the `wasmExecMsgs` array before they are executed.

Nibiru: Addressed by PR-2092

Zenith: Verified

[L-5] Missing fee validation in `SetParams` keeper

Severity: Low	Status: Resolved

Context:

- evm_state.go#L119

```
// SetParams: Setter for the module parameters.
func (k Keeper) SetParams(ctx sdk.Context, params evm.Params) {
        k.EvmState.ModuleParams.Set(ctx, params)
}
```

Description: The `SetParams` function defined in the `x/evm/keeper/evm_state.go:119` lacks validation of the `CreateFuntokenFee` parameter. This function directly sets the provided fee parameter without checking if it's within acceptable bounds. This function is invoked both during the genesis initialization through `InitGenesis` and when updating parameters via the `UpdateParams` function.

The absence of validation allows for the possibility of setting the creation fee for funtokens to values that are either excessively high or too low. An unreasonably low fee could lead to spam attacks possibility, where malicious actors flood the network with funtoken creation requests, overwhelming system resources and potentially degrading network performance. Conversely, an excessively high fee could deter legitimate users from creating funtokens at all.

Recommendation: We recommend implementing thorough validation within the `SetParams` ensuring that the `CreateFuntokenFee` parameter falls within predefined acceptable ranges. This can be achieved by defining minimum and maximum threshold values that represent reasonable fee limits.

Nibiru: Addressed by PR-2091

Zenith: Verified

[L-6] Unsupported weird ERC20 tokens

Severity: Low	Status: Acknowledged

Context:

- msg_server.go#L653

```
// Check expected Receiver balance after transfer execution
        recipientBalanceAfter, err := k.ERC20().BalanceOf(erc20Addr,
recipient, ctx)
        if err != nil {
                return nil, errors.Wrap(err, "failed to retrieve
balance")
        }
        if recipientBalanceAfter == nil {
                return nil, fmt.Errorf("failed to retrieve balance,
balance is nil")
        }

        expectedFinalBalance := big.NewInt(0).Add(recipientBalanceBefore,
coin.Amount.BigInt())
        if r := recipientBalanceAfter.Cmp(expectedFinalBalance); r != 0 {
                return nil, fmt.Errorf("expected balance after transfer
to be %s, got %s", expectedFinalBalance, recipientBalanceAfter)
        }
```

Description: The current implementation lacks support for certain non-standard ERC20 tokens, specifically those with "Fee on Transfer" mechanisms and those that have missing return values in their `transfer` functions.

According to the project documentation, these tokens should be properly handled by the system. The issue arises because fee-on-transfer tokens deduct a fee during transfer operations, meaning the recipient receives less than the amount specified. The fee is usually burned or redistributed according to the token's smart contract logic.

The existing code assumes however, that the actual transferred amount is equal to the intended transfer amount, so assertions or checks enforcing this equality fail when interacting with fee-on-transfer tokens, leading to transaction failures or incorrect accounting of token balances.

Similarly, some ERC20 tokens do not return a boolean value upon executing transfer or `transferFrom` functions, even though the ERC20 standard specifies that these functions should return true on success.

In the `x/evm/keeper/erc20.go:73`, the `Transfer` function, the code attempts to unpack the return value of the `transfer` function. If a token does not return any value, this unpacking operation fails, causing the transaction to revert and making it impossible to interact with such tokens through the current implementation.

There is an existing issue documented in NibiruChain/nibiru#2063 acknowledging that some tokens do not adhere strictly to the ERC20 standard. While tokens that lack fundamental ERC20 functions (like no `transfer` function) are rightly considered non-ERC20 and can be excluded, tokens with minor deviations like fee-on-transfer or missing return values are prevalent and widely used.

Recommendation: We recommend that to improve compatibility and user experience, the system should be updated to handle these "weird" ERC20 token behaviors gracefully. For fee-on-transfer tokens, the `transfer` logic should be adjusted to not assume that the transfer amount equals the received amount. After performing a `transfer`, the recipient's balance before and after the transfer should be retrieved to calculate the actual amount received, ensuring that any accounting or state updates reflect the actual amount received.

For tokens with missing return values, function calls should be modified to use low-level calls that do not expect a return value, avoiding unpacking the return value when calling the transfer function. After the transfer call, it's important to verify that the transaction did not revert by checking for errors or ensuring the transaction receipt indicates success.

Additionally, documentation and user guidance should be updated to inform users about token behaviors, providing guidelines for interacting with such tokens and advising on best practices.

Nibiru: Acknowledged

4.5 Informational

A total of 3 informational findings were identified.

[I-1] Improve bank coin metadata for ERC-20 funtokens

Severity: Informational	Status: Resolved

Context:

- x/evm/keeper/funtoken_from_erc20.go#L118-L132

Description: In `createFunTokenFromERC20()`, the bank metadata for the new funtoken coin is created with a single `DenomUnit`, with `Denom` set to `"erc20/{erc20}"` and `Exponent` set to `0`.

This can be improved in the following way by using two `DenomUnits`:

1. The smallest possible unit with `Exponent = 0`, universally called `wei` (i.e. `Denom = "wei"`) for all such ERC-20 funtokens
2. The regular unit which uses the ERC-20 token's decimals, i.e. `{ Denom: bankDenom, Exponent: info.Decimals }`

This first `wei` unit will be used as the coin's `Base` and the second unit as `Display`.

Additionally, the metadata `Name` could be set to the retrieved ERC-20 metadata `info.Name`.

Recommendation: Consider using two `DenomUnits` for the new funtoken coin metadata and setting the `Name` to the ERC-20 token's name. This would bring parity between EVM wallets, such as MetaMask, and Cosmos wallets, such as Keplr/Leap when displaying the coins/tokens.

Nibiru: Implemented with PR-2074

Zenith: Verified.

[I-2] Multiple functions are not used in the protocol's logic

Severity: Informational	Status: Acknowledged

Context:

- const.go#L131
- evm_state.go#L106
- logs.go#L15
- logs.go#L23
- logs.go#L31
- logs.go#L51
- logs.go#L56
- msg.go#L331
- msg.go#L420
- msg.go#L426
- params.go#L75
- statedb.go#L102
- statedb.go#L179
- statedb.go#L297
- statedb.go#L305
- statedb.go#L341
- statedb.go#L414
- statedb.go#L419
- statedb.go#L424
- statedb.go#L432
- statedb.go#L482

Description: It was found that the codebase contains multiple functions that, despite their implementation, and often also comments explaining the logic - are not used anywhere in the solution logic, and are also not externally callable. Most often, they are only called by individual unit tests, which in the context of blockchain is a suboptimal solution.

Furthermore, their existence increases the amount of the code as a whole, thus increasing the level of advancement for users familiarizing themselves with the source code and logic.

Recommendation: We recommend analyzing the possibility of removing the indicated functions, or moving them to the part of the code responsible for implementing helpers for unit tests.

Nibiru: Acknowledged

[I-3] Outdated comment and improper error message

Severity: Informational	Status: Resolved

Context:

- evmante_gas_consume.go#L162

```
// deductFee checks if the fee payer has enough funds to pay for the fees
and deducts them.
// If the spendable balance is not enough, it tries to claim enough
staking rewards to cover the fees.
func (anteDec AnteDecEthGasConsume) deductFee(
        ctx sdk.Context, fees sdk.Coins, feePayer sdk.AccAddress,
)
```

- precompile/oracle.go#L116

```
if len(args) != 1 {
                err = fmt.Errorf("expected 3 arguments but got %d",
len(args))
                return
        }
```

Description: In the `app/evmante/evmante_gas_consume.go:161`, there is a comment indicating that if a user's balance is insufficient to cover transaction fees, the system will attempt to claim enough staking rewards to cover the deficit:

```
// If the spendable balance is not enough, try to claim enough staking
rewards to cover the fees.
```

However, this functionality is not implemented in the code that follows. The `deductFee` function only attempts to deduct fees directly from the user's balance through a direct transfer of funds to the module account:

There is no logic present to claim staking rewards or to handle situations where the user's balance is insufficient. This discrepancy suggests that the comment was copied from the Evmos codebase, where such a feature is implemented within the `deductFee` logic, but the actual implementation was not carried over.

Another issue can be found in `x/evm/precompile/oracle.go:116`. The `decomposeQueryExchangeRateArgs` function verifies, if length of `args` parameter differs from 1, and if so - an error is returned.

However, through the error message the `"expected 3 arguments but got %d"` is returned, what is not aligned with the check mentioned above.

Recommendation:

1. We recommended to either implement the functionality as described in the comment or to update the comment to accurately reflect the current code behavior. Implementing the feature would involve adding logic to the `deductFee` function that checks if the user's spendable balance is insufficient to cover the fees and, if so, attempts to claim enough staking rewards to make up the difference.
2. For the `decomposeQueryExchangeRateArgs` function, we recommend changing the error message to indicate that only one argument was expected.

Nibiru: Resolved with PR-2088

Zenith: Verified