code4rena

Initia

Smart Contract Security Assessment

Audit dates: Feb 10 — Feb 24, 2025

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Initia Cosmos smart contract system. The audit took place from February 10 to February 24, 2025.

This audit was judged by LSDan.

Final report assembled by Code4rena.

Following the C4 audit, 4 wardens (berndartmueller, Franfran and a_kalout and ali_shehab of team 0xAlix2) reviewed the mitigations implemented by the Initia team; the mitigation review report is appended below the audit report.

Summary

The C4 analysis yielded an aggregated total of 16 unique vulnerabilities. Of these vulnerabilities, 8 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 7 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Initia Cosmos repository, and is composed of 14 smart contracts written in the Cosmos and Golang programming language.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling

- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (8)

[H-01] Wrong handling of ERC20 denoms in `ERC20Keeper::BurnCoins`

Submitted by OxAlix2

MiniEVM is an optimistic rollup consumer chain powered by EVM, it allows users to interact with EVM/solidity contracts through Cosmos. Part of those contracts is ERC20, which allows users to interact with `ERC20Factory` to create their ERC20s through Cosmos call messages. When doing so, a new ERC20 is created, and of course, an address is generated say `ADDRESS`, on the Cosmos end this denom/token is saved as `evm/ADDRESS` (without the `0x` prefix), and the corresponding EVM address is saved, this type of denoms are referred to as ERC20Denom. Another type of denoms created is through `CreateERC20`, these also interact with the factory, but these are saved as just denoms, in other words, if a `uinit` denom is created it is just saved and referred to as `uinit`, the Cosmos part handles the part of minting and burning of these.

When `MintCoins` is called in the ERC20 keeper, it checks if the denom is `ERC20Denom`, if so, nothing is done, this is because minting should be done by the ERC20 creator/owner. If not, then `sudoMint` is called, here:

```
func (k ERC20Keeper) MintCoins(ctx context.Context, addr sdk.AccAddress,
amount sdk.Coins) error {
        // ... snip ...

        for _, coin := range amount {
                denom := coin.Denom
                if types.IsERC20Denom(denom) {
                        return moderrors.Wrapf(types.ErrInvalidRequest,
"cannot mint erc20 coin: %s", coin.Denom)
                }

                // ... snip ...

                inputBz, err := k.ERC20ABI.Pack("sudoMint", evmAddr,
coin.Amount.BigInt())
                if err != nil {
```

```
                        return types.ErrFailedToPackABI.Wrap(err.Error())
                }

                // ... snip ...
        }

        // ... snip ...
}
```

Similarly, in `BurnCoins` (here):

```
func (k ERC20Keeper) BurnCoins(ctx context.Context, addr sdk.AccAddress,
amount sdk.Coins) error {
        // ... snip ...

        for _, coin := range amount {
                // if a coin is not generated from 0x1, then send the
coin to community pool
                // because we don't have burn capability.
                if types.IsERC20Denom(coin.Denom) {
                        if err :=
k.communityPoolKeeper.FundCommunityPool(ctx, amount, evmAddr.Bytes());
err != nil {
                                return err
                        }

                        continue
                }

                // ... snip ...
        }

        // ... snip ...
}
```

If the denom is `ERC20Denom`, it is supposed to be sent to the community pool. However, there's a huge flaw here, as it is sending the whole `amount` instead of `coin`.

If one of the coins returns true for `types.IsERC20Denom(coin.Denom)`, then all the coins are sent to `FundCommunityPool`, not just the matching one. Then, the next iteration will send the coins again to 0x1 to be burned. If 2 EVM coins are provided, then double of each will be sent to the fund.

Proof of Concept

Add the following test in `minievm/x/evm/keeper/erc20_test.go`:

```
func Test_BurnTwice(t *testing.T) {
        ctx, input := createDefaultTestInput(t)
        _, _, addr := keyPubAddr()
        evmAddr := common.BytesToAddress(addr.Bytes())

        erc20Keeper, err := keeper.NewERC20Keeper(&input.EVMKeeper)
        require.NoError(t, err)

        fooContractAddr_1 := deployERC20(t, ctx, input, evmAddr, "foo")
        fooDenom_1, err := types.ContractAddrToDenom(ctx,
&input.EVMKeeper, fooContractAddr_1)
        require.NoError(t, err)
        require.Equal(t, "evm/"+fooContractAddr_1.Hex()[2:], fooDenom_1)

        fooContractAddr_2 := deployERC20(t, ctx, input, evmAddr, "bar")
        fooDenom_2, err := types.ContractAddrToDenom(ctx,
&input.EVMKeeper, fooContractAddr_2)
        require.NoError(t, err)
        require.Equal(t, "evm/"+fooContractAddr_2.Hex()[2:], fooDenom_2)

        cacheCtx, _ := ctx.CacheContext()
        err = erc20Keeper.MintCoins(cacheCtx, addr, sdk.NewCoins(
                sdk.NewCoin(fooDenom_1, math.NewInt(100)),
                sdk.NewCoin(fooDenom_2, math.NewInt(100)),
        ))
        require.Error(t, err)

        mintERC20(t, ctx, input, evmAddr, evmAddr,
sdk.NewCoin(fooDenom_1, math.NewInt(100)), false)
        mintERC20(t, ctx, input, evmAddr, evmAddr,
sdk.NewCoin(fooDenom_2, math.NewInt(100)), false)

        amount, err := erc20Keeper.GetBalance(ctx, addr, fooDenom_1)
        require.NoError(t, err)
        require.Equal(t, math.NewInt(100), amount)

        amount, err = erc20Keeper.GetBalance(ctx, addr, fooDenom_2)
        require.NoError(t, err)
        require.Equal(t, math.NewInt(100), amount)

        // Community pool should have 0 balance
        require.Equal(t, math.NewInt(0),
input.CommunityPoolKeeper.CommunityPool.AmountOf(fooDenom_1))
```

```
        require.Equal(t, math.NewInt(0),
input.CommunityPoolKeeper.CommunityPool.AmountOf(fooDenom_2))

        // Burn 50 coins of each denom
        err = erc20Keeper.BurnCoins(ctx, addr, sdk.NewCoins(
                sdk.NewCoin(fooDenom_1, math.NewInt(50)),
                sdk.NewCoin(fooDenom_2, math.NewInt(50)),
        ))
        require.NoError(t, err)

        // Community pool should have 100 (50*2) coins of each denom
        require.Equal(t, math.NewInt(100),
input.CommunityPoolKeeper.CommunityPool.AmountOf(fooDenom_1))
        require.Equal(t, math.NewInt(100),
input.CommunityPoolKeeper.CommunityPool.AmountOf(fooDenom_2))
}
```

Recommended mitigation steps

minievm/x/evm/keeper/erc20.go:

```
        // BurnCoins implements IERC20Keeper.
        func (k ERC20Keeper) BurnCoins(ctx context.Context, addr
sdk.AccAddress, amount sdk.Coins) error {
                // ... snip ...

                for _, coin := range amount {
                        // if a coin is not generated from 0x1, then send
the coin to community pool
                        // because we don't have burn capability.
                        if types.IsERC20Denom(coin.Denom) {
-                               if err :=
k.communityPoolKeeper.FundCommunityPool(ctx, amount, evmAddr.Bytes());
err != nil {
+                               if err :=
k.communityPoolKeeper.FundCommunityPool(ctx, sdk.NewCoins(coin),
evmAddr.Bytes()); err != nil {
                                        return err
                                }

                                continue
                        }

                        // ... snip ...
                }
```

```
                return nil
        }
```

leojin (Initia) confirmed and commented:

We fixed in this PR.

Initia mitigated:

PR addresses burn coins error.

Status: Mitigation confirmed. Full details in reports from berndartmueller, Franfran and 0xAlix2.

[H-02] A regular Cosmos SDK message can be disguised as an EVM transaction, causing `ListenFinalizeBlock` to error which prevents the block from being indexed

Submitted by berndartmueller

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/keeper/txutils.go#L321

Finding description and impact

The indexer's `ListenFinalizeBlock()` is called whenever a block is finalized from within `BaseApp::FinalizeBlock()` and processes the block, including the transactions.

https://github.com/cosmos/cosmos-sdk/blob/eb1a8e88a4ddf77bc2fe235fc07c57016b7386f0/baseapp/abci.go#L856-L864

```
func (app *BaseApp) FinalizeBlock(req *abci.RequestFinalizeBlock) (res
*abci.ResponseFinalizeBlock, err error) {
        defer func() {
                // call the streaming service hooks with the
FinalizeBlock messages
                for _, streamingListener := range
app.streamingManager.ABCIListeners {
                        if err :=
streamingListener.ListenFinalizeBlock(app.finalizeBlockState.Context(),
*req, *res); err != nil {
                                app.logger.Error("ListenFinalizeBlock
listening hook failed", "height", req.Height, "err", err)
                        }
```

```
                }
        }()
```

During looping the transactions, it extracts the logs and the contract address from the transaction result:

indexer/abci.go#L86-L91

```
86: // extract logs and contract address from tx results
87: ethLogs, contractAddr, err := extractLogsAndContractAddr(txStatus,
txResult.Data, ethTx.To() == nil)
88: if err != nil {
89:     e.logger.Error("failed to extract logs and contract address",
"err", err)
90:     return err
91: }
```

`extractLogsAndContractAddr()` then tries to unpack the transaction data into either `MsgCreateResponse` in line 28 or `MsgCallResponse` in line 37 by calling `unpackData()`.

If the message's type URL is not matching the expected types from the `MsgCreateResponse` or `MsgCallResponse`, it returns an error in line `61`.

indexer/utils.go#L47-L70

```
47: // unpackData extracts msg response from the data
48: func unpackData(data []byte, resp proto.Message) error {
49:     var txMsgData sdk.TxMsgData
50:     if err := proto.Unmarshal(data, &txMsgData); err != nil {
51:             return err
52:     }
53:
54:     if len(txMsgData.MsgResponses) == 0 {
55:             return sdkerrors.ErrLogic.Wrap("failed to unpack data;
got nil Msg response")
56:     }
57:
58:     msgResp := txMsgData.MsgResponses[0]
59:     expectedTypeUrl := sdk.MsgTypeURL(resp)
60:     if msgResp.TypeUrl != expectedTypeUrl {
61:             return fmt.Errorf("unexpected type URL; got: %s,
expected: %s", msgResp.TypeUrl, expectedTypeUrl)
62:     }
63:
64:     // Unpack the response
```

```
65:     if err := proto.Unmarshal(msgResp.Value, resp); err != nil {
66:             return err
67:     }
68:
69:     return nil
70: }
```

In this case, the error will be propagated up to the `ListenFinalizeBlock` function in line 87, which will then exit the loop and stops processing the block. Consequently, the block and its transactions is not indexed and thus not available when querying via the JSON-RPC. Additionally, pruning is also not run and the bloom filter is not created.

Now the question is how this can be exploited?

The goal is to disguise a Cosmos SDK message, that is not a `MsgCall` or `MsgCreate`, so that `extractLogsAndContractAddr()` and more specifically, `unpackData()` errors due to the mismatch of the type URL, as an EVM transaction.

This can be done by using any Cosmos SDK message and signing it as if it were a EVM message (`MsgCreate`, `MsgCall`), by using `SignMode_SIGN_MODE_ETHEREUM`.

As a result, `ConvertCosmosTxToEthereumTx()` will consider this transaction to be an EVM transaction, bypassing the check in `txutils.go#L250-L253`.

Subsequently, it will also bypass the `switch/case` in `txutils.go#L286-L321`, as the type URL does not match any of the expected types (e.g., `/minievm.evm.v1.MsgCall`). The results is a valid EVM transaction, with the notable difference that `txData.To = nil` and `txData.Data = nil`.

The transaction will be successfully executed, but when processing it in `ListenFinalizeBlock`, it will error out due to the mismatch of the type URL in `unpackData()`, as it was expecting `types.MsgCreateResponse`.

Proof of Concept

The test case `Test_ListenFinalizeBlock_Audit_Errors` demonstrates how an ETH transaction can be wrapped with a bank `MultiSend` message, so that `ListenFinalizeBlock` errors, preventing the block from being indexed.

Please note that `CustomConvertEthereumTxToCosmosTx` is a copy of the original function. It only differs by adding the `MultiSend` Cosmos message instead of `MsgCreate` or `MsgCall`. It makes the PoC simpler. But it would also be possible to build such an transaction without `CustomConvertEthereumTxToCosmosTx`, using just the Cosmos tx builder.

Also, the `MultiSend` message handler was changed to not error. Again, also for simplicity. Any other Cosmos SDK message (e.g. `Send`) would work as well.

▶ Details

Recommended mitigation steps

I think it should be sufficient to fix this issue by adding a `default` case to the `switch` statement in `ConvertCosmosTxToEthereumTx()`, which returns `nil, nil, nil` so that such a transaction is not considered an EVM transaction.

beer-1 (Initia) disputed and commented:

If you change the message and keep the signature, then your tx cannot be entered to mempool because signature verification failed in ante level.

berndartmueller (warden) commented:

If you change the message and keep the signature, then your tx cannot be entered to mempool because signature verification failed in ante level.

There's a misunderstanding. The PoC does not change/replace the message. It simply uses `MsgMultiSend` instead of e.g. `MsgCall`. The signature remains valid. It's the same logic that `ConvertCosmosTxToEthereumTx` uses.

I noticed that the git diff contained an error so that it was not possible to `git apply` it. I've fixed it below. Also, I've added `fmt.Println` statements to the signature verification to show that the signature is indeed valid.

Please run the PoC to convince yourself of this issue. If you need further help of if something is unclear, please let me know. Happy to help!

▶ Details

LSDan (judge) commented:

Thank you for the additional detail. Reinstated the validity.

beer-1 (Initia) commented:

Still, I don't think this is possible at all. The problem you are saying is the second tx will return error when we call `ConvertCosmosTxToEthereumTx` function here.

This tx cannot enter this indexer code due to ante handler is already filter this type of transactions; see here. We are using skip-mev's block-sdk which is ensuring to create a block only with the txs those are valid in the ante handler.

berndartmueller (warden) commented:

I'm saying that `extractLogsAndContractAddr` errors. It will not be filtered in the ante handler. Please run the PoC to get a better feeling for the issue.

Initia mitigated:

PR here: `ConvertCosmosTxToEthereumTx` to properly check type url.

Status: Mitigation confirmed. Full details in reports from berndartmueller, Franfran and 0xAlix2.

[H-03] `ExecuteRequest`'s are not properly removed from the context queue

Submitted by berndartmueller

The `minievm` cosmos precompile allows a Solidity contract to dispatch a Cosmos SDK message, which is executed after the EVM call is successfully executed.

This is done by calling the cosmos precompile with the `execute_cosmos` or `execute_cosmos_with_options` function selector and passing the encoded message. This will wrap the message with `ExecuteRequest` and append it to the messages slice in the context with the key `types.CONTEXT_KEY_EXECUTE_REQUESTS`.

minievm:x/evm/precompiles/cosmos/contract.go#L287-L294

```
287: messages := ctx.Value(types.CONTEXT_KEY_EXECUTE_REQUESTS).(*
[]types.ExecuteRequest)
288: *messages = append(*messages, types.ExecuteRequest{
289:    Caller: caller,
290:    Msg:    sdkMsg,
291:
292:    AllowFailure: executeCosmosArguments.Options.AllowFailure,
293:    CallbackId:   executeCosmosArguments.Options.CallbackId,
294: })
```

Then, after the EVM call finished successfully in `EVMCallWithTracer()`, it executes the scheduled messages. The same is done in `EVMCreateWithTracer()`.

```
347: // handle cosmos execute requests
348: requests := sdkCtx.Value(types.CONTEXT_KEY_EXECUTE_REQUESTS).(*
[]types.ExecuteRequest)
349: if dispatchLogs, err := k.dispatchMessages(sdkCtx, *requests); err
!= nil {
350:    return nil, nil, err
351: } else {
352:    logs = append(logs, dispatchLogs...)
353: }
```

The issue is that dispatched messages are not removed from the queue in the scenario where they have been dispatched by an external Solidity that reverts but where a `try/catch` contains the error. In this case, the messages are still executed.

It seems this is because `prepareSDKContext(..)` sets the empty cosmos messages queue to the outer context, while any internal EVM that creates a new `statedb` snapshot will not result in a new context with it's own, empty `types.CONTEXT_KEY_EXECUTE_REQUESTS` queue. When the cosmos precompile retrieves the queue from the context with `ctx.Value(types.CONTEXT_KEY_EXECUTE_REQUESTS)`, the context is traversed up to the outer context which holds the key/value (pointer).

Overall, this is very problematic as it leads to unexpected behavior. For example, the external call reverts on purpose due to an invariant being violated, but the dispatched message that is still executed might be a sensitive operation (e.g., IBC token transfer) that should only be executed in the successful case. This can have severe consequences, including funds being locked or lost.

Proof of Concept

The following Poc amends the `Counter` Solidity contract by adding a new external function `nested(..)`, which dispatches a nested `MsgCall` EVM Cosmos SDK messages, followed by a `revert()`.

The `nested(..)` function is called in the `recursive(..)` function, wrapped in a `try/catch` block so that the revert is caught and does not propagate.

```
diff --git a/x/evm/contracts/counter/Counter.sol
b/x/evm/contracts/counter/Counter.sol
index 2c5166e..d6a4930 100644
--- a/x/evm/contracts/counter/Counter.sol
+++ b/x/evm/contracts/counter/Counter.sol
@@ -81,10 +81,18 @@ contract Counter is IIBCAsyncCallback {
             return;
         }

-        COSMOS_CONTRACT.execute_cosmos(_recursive(n));

-        // to test branching
+        try this.nested(n) {
+            // do nothing
+        } catch {
+            // do nothing
+        }
+    }
+
+    function nested(uint64 n) external {
         COSMOS_CONTRACT.execute_cosmos(_recursive(n));
+
+        revert();
```

```
    }

    function _recursive(uint64 n) internal returns (string memory
message) {
```

After applying the git diff, re-compile the Solidity contracts with `make contracts-gen`.

Then, copy and paste the following test case in `x/evm/keeper/context_test.go`. It will show that the dispatched message is still executed, by verifying that the `logs` contain the `recursive_called` event twice.

```
func Test_Recursive_Audit_ExecuteRequestsNotCleanedOnRevert(t *testing.T)
{
        ctx, input := createDefaultTestInput(t)
        _, _, addr := keyPubAddr()

        counterBz, err := hexutil.Decode(counter.CounterBin)
        require.NoError(t, err)

        // deploy counter contract
        caller := common.BytesToAddress(addr.Bytes())
        retBz, contractAddr, _, err := input.EVMKeeper.EVMCreate(ctx,
caller, counterBz, nil, nil)
        require.NoError(t, err)
        require.NotEmpty(t, retBz)
        require.Len(t, contractAddr, 20)

        // call recursive function
        parsed, err := counter.CounterMetaData.GetAbi()
        require.NoError(t, err)

        inputBz, err := parsed.Pack("recursive", uint64(1))
        require.NoError(t, err)

        _, logs, err := input.EVMKeeper.EVMCall(ctx, caller,
contractAddr, inputBz, nil, nil)
        require.NoError(t, err)
        require.Len(t, logs, 2)

        // check logs
        log := logs[0]
        require.Equal(t, contractAddr.Hex(), log.Address)
        require.Equal(t, parsed.Events["recursive_called"].ID.Hex(),
log.Topics[0])
```

```
        require.Equal(t,
"0x0000000000000000000000000000000000000000000000000000000000000001",
log.Data)

        log = logs[1]
        require.Equal(t, contractAddr.Hex(), log.Address)
        require.Equal(t, parsed.Events["recursive_called"].ID.Hex(),
log.Topics[0])
        require.Equal(t,
"0x0000000000000000000000000000000000000000000000000000000000000000",
log.Data)
}
```

Recommended mitigation steps

Consider making sure that the `ExecuteRequest`'s that have been dispatched by an EVM call that reverted, are properly removed and not executed.

beer-1 (Initia) confirmed and commented:

Here is the fix.

Initia mitigated:

Fixed to maintain execute request on snapshot.

Status: Mitigation confirmed. Full details in reports from 0xAlix2, berndartmueller and Franfran.

[H-04] JSON-RPC `FilterCriteria.Addresses` are unbound and can be used to DoS the RPC

Submitted by berndartmueller

The `eth_getLogs` ETH JSON-RPC method returns an array of all logs matching a given filter criteria, e.g., a topic list `Topics` and/or a list of addresses (`Addresses`) that restricts matches to events created by specific contracts.

While `Topics` is limited to `maxTopics = 4`, `Addresses` in unbounded.

minievm:jsonrpc/namespaces/eth/filters/api.go#L366

```
358: // GetLogs returns logs matching the given argument that are stored
within the state.
359: func (api *FilterAPI) GetLogs(ctx context.Context, crit
ethfilters.FilterCriteria) ([]*coretypes.Log, error) {
360:     if len(crit.Topics) > maxTopics {
361:             return nil, errExceedMaxTopics
```

```
362:     }
363:     var filter *Filter
364:     if crit.BlockHash != nil {
365:             // Block filter requested, construct a single-shot filter
366: ❌  filter = newBlockFilter(api.logger, api.backend, *crit.BlockHash,
crit.Addresses, crit.Topics)
367:     } else {
```

This can be exploited to DoS the RPC server and bring it to a halt by sending `eth_getLogs` requests with a large number of addresses in the `Addresses` field, which takes a considerable amount of time to process.

It should be noted that the JSON-RPC server has a default request body limit of `~10MB`, which can be adjusted via the `max-recv-msg-size` **configuration option**. This at least limits the `Addresses` field to a certain extent. However, the below the PoC demonstrates that this is not sufficient and it is still possible to DoS the server.

The same issue, not having an upper bound on the `Addresses`, is also observed in:

- `FilterAPI::NewFilter()` and
- `FilterAPI::Logs()`

Proof of Concept

The following estimation shows how many addresses we can fit within a `~10MB` JSON-RPC request limit:

1. Each Ethereum address is 20 bytes
2. In JSON-RPC format, each address needs to be hex-encoded with "0x" prefix:
    - 20 bytes becomes 40 hex characters
    - Plus "0x" prefix = 42 characters
3. In JSON array format, we need:
    - Quotes around each address: +2 characters = 44 chars per address
    - Commas between addresses: +1 character
    - Square brackets: +2 characters for the whole array
4. Each character in JSON is 1 byte

So for `n` addresses:

- Total bytes = `44n + (n-1) + 2`
- Simplified: `45n + 1 bytes`

With a 10MB (10,485,760 bytes) limit:

```
10,485,760 = 45n + 1
n = (10,485,760 - 1) / 45
```

```
n ≈ 232,905 addresses
```

To account for some request overhead, a more conservative estimate would be around **200,000** addresses.

The following test case demonstrates that by providing 200,000 addresses to the `GetLogs` function, the request takes more than 5 seconds to complete. This opens up the possibility of a DoS attack by sending a large number of addresses to the `GetLogs` function and repeating the request to bring the RPC to a halt.

Apply the following git diff and run the `Test_GetLogs` test with `go test -timeout 30s -run ^Test_GetLogs$ ./jsonrpc/namespaces/eth/filters`

```
diff --git a/jsonrpc/namespaces/eth/filters/api_test.go
b/jsonrpc/namespaces/eth/filters/api_test.go
index e1ad746..7c27536 100644
--- a/jsonrpc/namespaces/eth/filters/api_test.go
+++ b/jsonrpc/namespaces/eth/filters/api_test.go
@@ -392,10 +392,27 @@ func Test_GetLogs(t *testing.T) {
                }
        }

+       // create a massive number of addresses and populate them into
`Addresses`
+       numAddresses := 200_000
+       addresses := make([]common.Address, 0, numAddresses)
+       for i := 0; i < numAddresses; i++ {
+               addresses = append(addresses,
common.BytesToAddress(contractAddr))
+       }
+
+       // time how long it takes
+       start := time.Now()
+
        logs, err := input.filterAPI.GetLogs(context.Background(),
ethfilters.FilterCriteria{
                FromBlock: big.NewInt(tx2Height),
+               Addresses: addresses,
        })
        require.NoError(t, err)
+
+       elapsed := time.Since(start)
+
+       // assert that it takes at least 5 seconds
+       require.GreaterOrEqual(t, elapsed.Milliseconds(), int64(5_000))
```

```
+
        for _, log := range logs {
                if log.BlockNumber == uint64(tx3Height) {
                        require.Equal(t, txHash3, log.TxHash)
```

Recommended mitigation steps

Consider enforcing an upper limit on the number of `Addresses`, similar to how it's done for `Topics`.

beer-1 (Initia) confirmed and commented:

This is valid finding, but only consensus breaking and taking over funding related findings can be critical.

Would like to lower severity to medium.

LSDan (judge) commented:

I disagree. This is a direct DDOS vector with next to zero cost. The ease of execution and the potential for overloading every JSON-RPC server simultaneously with a small botnet is enough to justify high risk.

beer-1 (Initia) commented:

Fixed here.

Initia mitigated:

Added a max addresses limit.

Status: Mitigation confirmed. Full details in reports from 0xAlix2, berndartmueller and Franfran.

[H-05] Minievm fails to charge intrinsic gas costs for EVM transactions, allowing the abuse of the accesslist to consume computational resources without proper compensation

Submitted by berndartmueller

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/keeper/context.go#L372-L385

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/keeper/context.go#L243-L252

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/keeper/context.go#L277-L290

Finding description and impact

The minievm fails to charge intrinsic gas costs for EVM transactions in `EVMStaticCallWithTracer()`, `EVMCallWithTracer()`, and `EVMCreateWithTracer()`. In the EVM, intrinsic gas costs are considered costs for e.g., the accesslist (EIP-2930). For comparison, Evmos charges the intrinsic gas here.

According to the comment in `x/evm/state/statedb.go#L610`, EIP-2930 (access lists) is supposed to be supported. Overall, this represents a significant DoS risk to the network, as it allows for the potential abuse of computational resources without proper compensation.

Proof of Concept

Per EIP-2930, the Berlin hard fork raised the "cold" cost of account access opcodes (such as `BALANCE`, all `CALL`(s), and `EXT*`) to 2600 and raised the "cold" cost of state access opcode (`SLOAD`) from 800 to 2100 while lowering the "warm" cost for both to 100.

However, EIP-2930 has the added benefit of lowering transaction costs due to the transaction's 200 gas discount.

As a result, instead of paying 2600 and 2100 gas for a `CALL` and `SLOAD` respectively, the transaction only requires 2400 and 1900 gas for cold access, and subsequent warm access will only cost 100 gas.

In the EVM, `makeCallVariantGasCallEIP2929` checks if the address is warm. If so, the gas costs are reduced. This is under the premise that the intrinsic gas costs have been paid upfront, which include the accesslist costs.

However, as the intrinsic gas is not charged at all in `minievm`, neither the regular 2600 and 2100 nor the discounted 2400 and 1900 gas for the `CALL` and `SLOAD` opcodes is paid for.

By providing a large number of distinct contract addresses in the accesslist, the user does not properly pay for the computational resources required to load the code from storage. Similar with `SLOAD`'s.

This is additionally magnified by the fact that the access list is iterated on-chain when processing the message, utilizing uncapped computational resources which the user does not explicitly pay for.

Recommended mitigation steps

Consider charging intrinsic transaction gas, that considers the accesslist, as per EIP-2930.

leojin (Initia) confirmed and commented:

We fixed the intrinsic gas charge in this PR. But, since gas is charged on the cosmos side based on the transaction's data size (like intrinsic gas do), it does not pose a significant DoS risk; therefore, its security level can be considered lower.

berndartmueller (warden) commented:

It's correct that the `ConsumeTxSizeGasDecorator` ante decorator charges gas per tx byte:

```
ctx.GasMeter().ConsumeGas(params.TxSizeCostPerByte*storetypes.Gas(len(ctx
.TxBytes())), "txSize")
```

where `TxSizeCostPerByte = 10` by default.

However, this is, in my opinion, not sufficient. Considering that the access list `AccessList []AccessTuple` is basically an array of EVM addresses (`Address string`), which needs 20 bytes + a few bytes of overhead for the encoding per entry. To simplify calculations, let's assume that an address entry in the access list requires 30 bytes. This results in `30 * 10 = 300 gas` being charged by `ConsumeTxSizeGasDecorator(..)`.

However, this single access list entry equals 2600 of gas that would have been charged if the access list is considered in the intrinsic gas. That's a `2600 - 300 = 2300 gap`.

Hence, the gas that is charged per tx byte is not sufficient, leaving the DoS risk to be significant and justifying high severity.

LSDan (judge) commented:

Thank you for the clarified example. On review, I agree that this rises to the level of high risk.

Initia mitigated:

Fixed intrinsic gas charge.

Status: Mitigation confirmed. Full details in reports from 0xAlix2, berndartmueller and Franfran.

[H-06] Explicit gas limit on low-level Solidity calls can be bypassed by dispatched EVM calls via the custom Cosmos precompile

Submitted by berndartmueller

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/keeper/context.go#L556

Finding description and impact

It is possible to dispatch Cosmos SDK messages from within a Solidity contract, by using the custom cosmos precompile.

Those Cosmos SDK messages (e.g. `MsgCall`), specifically `ExecuteRequest`, are then dispatched and executed after the EVM call has successfully executed in `dispatchMessage()` via the message handler in line `556`.

```
491: func (k Keeper) dispatchMessage(parentCtx sdk.Context, request
types.ExecuteRequest) (logs types.Logs, err error) {
...     // [...]
547:
548:    // find the handler
549:    handler := k.msgRouter.Handler(msg)
550:    if handler == nil {
551:            err = types.ErrNotSupportedCosmosMessage
552:            return
553:    }
554:
555:    // and execute it
556:    res, err := handler(ctx, msg)
557:    if err != nil {
558:            return
559:    }
560:
561:    // emit events
562:    ctx.EventManager().EmitEvents(res.GetEvents())
563:
564:    // extract logs
565:    dispatchLogs, err := types.ExtractLogsFromResponse(res.Data,
sdk.MsgTypeURL(msg))
566:    if err != nil {
567:            return
568:    }
569:
570:    // append logs
571:    logs = append(logs, dispatchLogs...)
572:
573:    return
574: }
```

However, as we can see above in the code snippet, there is no new, local, gas meter created for the handler. Instead, the same gas meter as the parent call, i.e., the `MsgCall` that initiated the root EVM call, is used. Hence, the same gas limit is used for the dispatched message as for the parent call.

This can be exploited by a solidity contract that's called with a restricted gas limit (e.g. by using low-level call with `gas` set to 100k), which can use the `COSMOS_CONTRACT` precompile (at address `0xf1`) to schedule another EVM `MsgCall`. This EVM call (`MsgCall`) is then not restricted with the same gas limit as provided in the low-level call, rather it can use the full parent's gas limit.

Contrary to what a Solidity developer would expect when explicitly setting a gas limit on a low-level call, e.g., where the target contract is not fully trusted due to being an arbitrary external call.

Note that the same issue applies to the `callback` **call in the** `defer` **block** of the `dispatchMessage()` function. The callback EVM call also uses the same gas meter as the parent call.

Proof of Concept

The following git diff adds a test case `Test_CallChildContractBypassingGasLimit` to `context_test.go`, which demonstrates that a child contract that is called with an explicit gas limit of 100k via a low-level call can bypass this gas limit via the `CosmosPrecompile` precompile to dispatch an EVM call that is then executed with the initial gas limit (1M).

Specifically, the `nested()` function will emit `NestedCalled` with the `gasleft()` value, showing that it was called with almost the full 1M gas limit, instead of the 100k gas limit.

▶ Details
The following git diff:

- Adds a new `Untrusted` contract that is called by `Counter.recursive` with an explicit gas limit of 100k.
- Amends `Counter.recursive` to call the `Untrusted` contract with an explicit gas limit.

▶ Details
Make sure to run `make contracts-gen` to generate the Go bindings for the Solidity contracts before running the test.

Recommended mitigation steps

Consider providing the cosmos precompile's remaining gas as an additional parameter in `ExecuteRequest` when dispatching the request.

This would allow using this gas limit in `dispatchMessage()` to create a custom gas meter with the specific gas limit, ensuring that dispatched messages are limited to the intended gas limit.

beer-1 (Initia) confirmed and commented:

Fixed here.

Initia mitigated:

Introduced to use `submsg` gas limit.

Status: Unmitigated. Full details in reports from berndartmueller and 0xAlix2, and also included in the Mitigation Review section below.

[H-07] EVM stack overflow error leads to no gas being charged, which can be exploited to DoS the chain by dispatching EVM calls via the cosmos precompile

Submitted by berndartmueller

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/keeper/context.go#L300-L303

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/keeper/context.go#L405-L407

Finding description and impact

Both the `EVMCallWithTracer()` and `EVMCreateWithTracer()` functions check the special case where `gasRemaining == 0 && err != nil && err != vm.ErrOutOfGas`, which can happen in scenarios like:

- Stack overflow errors
- Invalid jump destinations
- Invalid opcodes

```
// evm sometimes return 0 gasRemaining, but it's not an out of gas error.
if gasRemaining == 0 && err != nil && err != vm.ErrOutOfGas {
        return nil, common.Address{}, nil,
types.ErrEVMCreateFailed.Wrap(err.Error())
}
```

In this situation, it returns early with an error, without adding the used EVM gas to the gas meter. Consequently, the EVM call was gas-free.

This if not a particular big problem for regular Cosmos SDK EVM calls, as the user already paid for the full gas (limit) upfront. The issue it causes is that the block gas meter will be incorrectly updated, it will not reflect the actual gas used by the EVM call; the impact of this is limited.

However, it is specifically problematic in the context of EVM calls that have been dispatched via the `CosmosPrecompile` precompile. Such calls are executed by `dispatchMessage()`,

which does not charge the gas upfront. Therefore, no gas is charged at all of such dispatched EVM calls, which allows DoS attacks by repeating such calls.

Proof of Concept

The following PoC demonstrates a stack overflow error in the `Counter` contract during a dispatched (nested) EVM call. It shows that in this case, no gas is charged for the nested call.

Calling `recursive()` will dispatch another EVM call to `Counter.nested()` (that is allowed to fail so that the error is not propagated), which then first consumes a lot of gas by calling `consumeGas(1_000_000)`, followed by triggering a stack overflow by calling `triggerStackOverflow()`.

In this case, `gasRemaining` will be 0, and `err != vm.ErrOutOfGas`, resulting in no gas being charged for the nested call, leaving the overall gas used below 100,000.

First, update the `Counter` contract:

```
diff --git a/x/evm/contracts/counter/Counter.sol
b/x/evm/contracts/counter/Counter.sol
index 2c5166e..6659e79 100644
--- a/x/evm/contracts/counter/Counter.sol
+++ b/x/evm/contracts/counter/Counter.sol
@@ -23,6 +23,27 @@ contract Counter is IIBCAsyncCallback {
         increase_for_fuzz(num - 1);
     }

+    function triggerStackOverflow() internal {
+        // Local variables to fill up the stack
+        uint256 a1 = 1;
+        uint256 a2 = 2;
+        uint256 a3 = 3;
+        uint256 a4 = 4;
+        uint256 a5 = 5;
+
+        // Recursive call that will overflow the stack
+        triggerStackOverflow();
+
+        // This line will never be reached
+        count += a1 + a2 + a3 + a4 + a5;
+    }
+
+    function consumeGas(uint256 iterations) internal {
+        for (uint256 i = 0; i < iterations; i++) {
+            count++;
+        }
```

```
+    }
+
     function increase() public payable {
         count++;
         emit increased(count - 1, count);
@@ -59,7 +80,7 @@ contract Counter is IIBCAsyncCallback {
         string memory exec_msg,
         bool allow_failure,
         uint64 callback_id
-    ) external {
+    ) public {
         COSMOS_CONTRACT.execute_cosmos_with_options(
             exec_msg,
             ICosmos.Options(allow_failure, callback_id)
@@ -77,14 +98,13 @@ contract Counter is IIBCAsyncCallback {
     function recursive(uint64 n) public {
         emit recursive_called(n);

-        if (n == 0) {
-            return;
-        }
+        execute_cosmos_with_options(_recursive(n), true, 0); //
dispatches an EVM call to `nested()`, allows call to fail, no callback
+    }

-        COSMOS_CONTRACT.execute_cosmos(_recursive(n));
+    function nested() external {
+        consumeGas(1_000_000);

-        // to test branching
-        COSMOS_CONTRACT.execute_cosmos(_recursive(n));
+        triggerStackOverflow();
     }

     function _recursive(uint64 n) internal returns (string memory
message) {
@@ -99,7 +119,7 @@ contract Counter is IIBCAsyncCallback {
                 '",',
                 '"input": "',
                 Strings.toHexString(
-                    abi.encodePacked(this.recursive.selector,
abi.encode(n - 1))
+                    abi.encodePacked(this.nested.selector)
                 ),
                 '",',
                 '"value": "0",',
```

Then, copy and paste the following test case to `x/evm/keeper/context_test.go`:

```
func Test_StackOverflowNoGasCharged(t *testing.T) {
        ctx, input := createDefaultTestInput(t)
        _, _, addr := keyPubAddr()

        counterBz, err := hexutil.Decode(counter.CounterBin)
        require.NoError(t, err)

        // deploy counter contract
        caller := common.BytesToAddress(addr.Bytes())
        retBz, contractAddr, _, err := input.EVMKeeper.EVMCreate(ctx,
caller, counterBz, nil, nil)
        require.NoError(t, err)
        require.NotEmpty(t, retBz)
        require.Len(t, contractAddr, 20)

        // call recursive function
        parsed, err := counter.CounterMetaData.GetAbi()
        require.NoError(t, err)

        inputBz, err := parsed.Pack("recursive", uint64(1))
        require.NoError(t, err)

        gasBefore := ctx.GasMeter().GasConsumed()

        _, logs, err := input.EVMKeeper.EVMCall(ctx, caller,
contractAddr, inputBz, nil, nil)

        gasUsed := ctx.GasMeter().GasConsumed() - gasBefore

        require.NoError(t, err)
        require.Equal(t, 1, len(logs))
        require.LessOrEqual(t, gasUsed, uint64(100_000))
}
```

Compile the Solidity contracts with `make contracts-gen` and run the test case.

Recommended mitigation steps

Consider consuming the gas before returning the error in lines `302` and `407` to always charge gas regardless of the error.

beer-1 (Initia) confirmed and commented:

We already received this review in other audit, but it is not publicly exposed during competition, so it should be valid report.

Fixed here.

Initia mitigated:

Changed to return exact error only at `simulate` or `checktx`.

Status: Mitigation confirmed. Full details in reports from 0xAlix2, berndartmueller and Franfran.

[H-08] Precompiles fail to charge gas in case of an error leading to a DOS vector

Submitted by berndartmueller

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/precompiles/cosmos/contract.go#L128-L139

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/precompiles/erc20_registry/contract.go#L74-L84

Finding description and impact

The custom precompiles in `minievm` are called via `ExtendedRun(..)` from the EVM.

Gas is charged based on the `input` length and a constant `GAS_PER_BYTE`. However, if either no ABI method is found or the input cannot be unpacked, the precompile returns an error in lines `84` or `89` without charging any gas.

In this case, it is possible to craft a transaction that can take a long time to execute, causing the expected block time to be exceeded and leading to a DOS vector for the network.

minievm:x/evm/precompiles/jsonutils/contract.go#L82-L93

```
55: func (e *JSONUtilsPrecompile) ExtendedRun(caller vm.ContractRef,
input []byte, suppliedGas uint64, readOnly bool) (resBz []byte, usedGas
uint64, err error) {
...     // [...]
81:
82:     method, err := e.ABI.MethodById(input)
83:     if err != nil {
84:             return nil, 0,
types.ErrPrecompileFailed.Wrap(err.Error())
85:     }
```

```
86:
87:     args, err := method.Inputs.Unpack(input[4:])
88:     if err != nil {
89:             return nil, 0,
types.ErrPrecompileFailed.Wrap(err.Error())
90:     }
91:
92:     // charge input gas
93:
ctx.GasMeter().ConsumeGas(storetypes.Gas(len(input))*GAS_PER_BYTE, "input
bytes")
```

According to the Initia docs, block times are planned to be 500ms.

Proof of Concept

The following PoC demonstrates how a very simplistic loop contract (bytecode returned by `getLoopBytecode()`) repeatedly `staticcall`s the jsonutils precompile with an unknown function selector (so that it does not charge gas that is relative to the input length in line `93`) until it runs out of gas. But by doing so, the execution time is significantly greater than the charged gas can compensate for.

Considering a 30M block gas limit, the loop contract execution can take `+2s` (on a maxed-out MacBook Pro M2)!

Please note that for simplicity, the loop contract is not regularly deployed, instead, `EVMCallWithTracer()` is modified to set the loop contract bytecode at the address `0x0200`. This does not affect the legitimacy of the PoC.

▶ Details

Running the test shows the execution time and gas used:

```
execution time: 2.138438458s
gas used: 30000000
```

Recommended mitigation steps

Consider charging some kind of static gas for each such precompile call, before erroring out. Or, charge the input gas before returning the error in lines `84` or `89`.

beer-1 (Initia) commented:

Seems go-ethereum is already charging the static cost here. This is more related with this issue.

This was fixed here.

LSDan (judge) commented:

This is more related with this issue.

Agreed, per Supreme Court decisions.

berndartmueller (warden) commented:

While this submission and S-151 are both talking about gas consumption issues, the submission here is unrelated. Considering that the provided fix mitigates S-151, it would not fix this issue here.

Here, it's basically that consuming gas in L93 (`ctx.GasMeter().ConsumeGas(storetypes.Gas(len(input))*GAS_PER_BYTE, "input bytes")`) is done after:

- Calling `MethodById()`, which looks up a method by the 4-byte selector.
- Calling `Unpack()`, which unmarshals the `[]byte` input into the corresponding `args` interface (e.g., `MergeJSONArguments`).

Both calls can error, in which the precompile returns early without charging the input gas.

Seems go-ethereum is already charging the static cost here.

`CallGasEIP150 = 700` is not actually charged. Since EIP-2929, `WARM_STORAGE_READ_COST = 100` is charged for warm (repeated) `STATICCALL` (and others).

Logging the gas cost for each `STATICCALL` iteration in the loop contract reveals that it costs 125 gas each round.

As seen in the PoC, it results in a potentially long-running tx execution (`~2.13s`). The static (constant) gas cost seems insufficient to prevent such scenarios.

Overall, I recommend following my recommendation to charge the input gas in the precompile before erroring. I would also appreciate if the judge could re-evaluate the duplication and eventually consider it a separate finding.

LSDan (judge) commented:

Thanks for the clarification. I agree that this makes sense as a separate issue.

Initia mitigated:

Changed to return exact error only at `simulate` or `checktx`.

Status: Unmitigated. Full details in reports from berndartmueller, 0xAlix2 and Franfran, and also included in the Mitigation Review section below.

Medium Risk Findings (8)

[M-01] `setBeforeSendHook` can never delete an existing store due to vulnerable validate

Submitted by oakcobalt, also found by 0xAlix2 and gxh191

https://github.com/initia-labs/miniwasm/blob/ed1728e2ad51f214ee0191dde62d96a074f55f1f/x/tokenfactory/types/msgs.go#L145-L148

Finding description

`setBeforeSendHook` allows disabling an existing token transfer hook (deleting a store) by setting an empty `cosmwasmAddress`. However, this functionality is not available due to a vulnerable `validate` flow.

Proof of Concept

We see the intention of allowing empty `cosmwasmAddress` to disable a hook in `before_send::setBeforeSendHook`. This is also consistent with osmosis token factory where empty `cosmwasmAddress` is a valid input to delete the current store.

```
//miniwasm/x/tokenfactory/keeper/before_send.go
func (k Keeper) setBeforeSendHook(ctx context.Context, denom string,
cosmwasmAddress string) error {
...
        // delete the store for denom prefix store when cosmwasm address
is nil
|>      if cosmwasmAddress == "" {
                return k.DenomHookAddr.Remove(ctx, denom) //@audit empty
cosmwasmAddress is used to delete the store
        } else {
                // if a contract is being set, call the contract using
cache context
                // to test if the contract is an existing, valid
contract.
                cacheCtx, _ := sdk.UnwrapSDKContext(ctx).CacheContext()
```

https://github.com/initia-labs/miniwasm/blob/ed1728e2ad51f214ee0191dde62d96a074f55f1f/x/tokenfactory/keeper/before_send.go#L26-L27

However, we see in `msg_server::SetBeforeSendHook` -> `msgs::validate`, empty `cosmwasmAddress` will always return error in `validate`.

```
//miniwasm/x/tokenfactory/keeper/msg_server.go
func (server msgServer) SetBeforeSendHook(ctx context.Context, msg
*types.MsgSetBeforeSendHook) (*types.MsgSetBeforeSendHookResponse, error)
{
        //@audit when msg.cosmwasmAddress is empty, Validate will always
return error.
|>      if err := msg.Validate(server.ac); err != nil {
                return nil, err
        }
...
```

https://github.com/initia-labs/miniwasm/blob/ed1728e2ad51f214ee0191dde62d96a074f55f1f/x/tokenfactory/keeper/msg_server.go#L198

```
//miniwasm/x/tokenfactory/types/msgs.go
func (m MsgSetBeforeSendHook) Validate(accAddrCodec address.Codec) error
{
...
        if addr, err := accAddrCodec.StringToBytes(m.CosmwasmAddress);
err != nil {
                return err
        } else if len(addr) == 0 {
                return ErrEmptySender
        }
...
```

https://github.com/initia-labs/miniwasm/blob/ed1728e2ad51f214ee0191dde62d96a074f55f1f/x/tokenfactory/types/msgs.go#L177-L180

Impact

Deleting an existing before send hook will always error out.

Recommended mitigation steps

In `func (m MsgSetBeforeSendHook) Validate`, add a bypass for `m.CosmwasmAddress == ""`.

hoon (Initia) confirmed and commented via duplicate issue S-18:

It was fixed in this commit. Since address validation is handled in `setBeforeSendHook`, it seems good to remove it.

Initia mitigated:

Delete `cosmwasm` address validation.

Status: Mitigation confirmed. Full details in reports from berndartmueller, 0xAlix2 and Franfran.

[M-02] Contract deployment restriction can be bypassed

Submitted by berndartmueller, also found by givn

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/keeper/context.go#L388-L401

Finding description and impact

The `Create` and `Create2` message handlers check the sender is allowed to deploy a contract by calling `assertAllowedPublishers(..)` here and here.

The deployers can either be restricted to specific addresses (`params.AllowedPublishers`) or completely unrestricted so that anyone can deploy a contract.

However, if the deployers are restricted, a permissioned deployer can deploy a factory contract that can then be used by anyone to deploy arbitrary contracts, bypassing the restriction and potentially leading to malicious contracts being deployed.

This is contrary to Evmos which does check also the EVM `create` opcode if the address is allowed to deploy a contract. It adds a special "create" hook, which checks if the caller is permissioned to deploy a contract.

This hook is called by `EVM::Create` and `EVM::Create2`, which are called when the EVM encounters the `CREATE` and `CREATE2` opcodes.

This shows that contract deployment can be fully restricted if desired.

Proof of Concept

`EVMCreateWithTracer()` internally calls the EVM's `Create` or `Create2` functions, which contain the standard implementation without any deployment restriction, found here.

Recommended mitigation steps

Consider using a similar approach to Evmos whereby the `Create` and `Create2` are intercepted and the caller is checked for permission to deploy a contract.

leojin (Initia) disputed and commented via duplicate issue S-161:

Since the allowed publisher list is typically set at chain launch, this does not seem to be an issue that requires fixing. It also appears fine since the update parameters are controlled by an operator.

[M-03] `COINBASE` opcode returns an empty address instead of the block proposer resulting in incompatibility with the EVM

Submitted by berndartmueller

EVM's `COINBASE` opcode (https://www.ethervm.io/#41) is supposed to return the block proposer. However, minievm sets `Coinbase` to an empty address in line `131`. This causes incompatible behavior with the EVM and potential issues with Solidity contracts that might use the coinbase address.

Proof of Concept

minievm:x/evm/keeper/context.go#L131

```
058: func (k Keeper) buildBlockContext(ctx context.Context,
defaultBlockCtx vm.BlockContext, evm *vm.EVM, fee types.Fee)
(vm.BlockContext, error) {
059:     sdkCtx := sdk.UnwrapSDKContext(ctx)
060:     baseFee, err := k.baseFee(ctx, fee)
061:     if err != nil {
062:             return vm.BlockContext{}, err
063:     }
064:
065:     return vm.BlockContext{
...:             // [...]
131:❌           Coinbase:    common.Address{},
132:             Difficulty:  big.NewInt(0),
133:             BlobBaseFee: big.NewInt(0),
134:     }, nil
135: }
```

The `GASLIMIT` EVM opcode uses this `GasLimit` value to return the block gas limit:

core/vm/instructions.go#L473-L476

```
func opCoinbase(pc *uint64, interpreter *EVMInterpreter, scope
*ScopeContext) ([]byte, error) {

scope.Stack.push(new(uint256.Int).SetBytes(interpreter.evm.Context.Coinba
se.Bytes()))
```

```
        return nil, nil
}
```

Recommended mitigation steps

Consider setting `Coinbase` to `stakingKeeper.GetValidatorByConsAddr()`, similar to how Evmos does it here and here.

leojin (Initia) disputed and commented:

Its intended spec to provide zero address.

[M-04] `GASLIMIT` opcode returns transaction gas limit instead of block gas limit resulting in incompatibility with the EVM

Submitted by berndartmueller

The EVM `GASLIMIT` opcode (https://www.ethervm.io/#45) is supposed to return the **block** gas limit.

However, when building the `BlockContext` in `buildBlockContext()`, `GasLimit` is set to the current transaction's remaining gas, which is calculated as `sdkCtx.GasMeter().Limit() - sdkCtx.GasMeter().GasConsumedToLimit()`. This results in incompatibility with the EVM and potential issues with Solidity contracts.

For comparison, Evmos correctly uses the block gas meter.

Proof of Concept

minievm:x/evm/keeper/context.go#L70

`GasLimit` is determined by calling `computeGasLimit()`.

```
58: func (k Keeper) buildBlockContext(ctx context.Context,
defaultBlockCtx vm.BlockContext, evm *vm.EVM, fee types.Fee)
(vm.BlockContext, error) {
59:     sdkCtx := sdk.UnwrapSDKContext(ctx)
60:     baseFee, err := k.baseFee(ctx, fee)
61:     if err != nil {
62:             return vm.BlockContext{}, err
63:     }
64:
65:     return vm.BlockContext{
66:             BlockNumber: defaultBlockCtx.BlockNumber,
67:             Time:        defaultBlockCtx.Time,
68:             Random:      defaultBlockCtx.Random,
69:             BaseFee:     baseFee,
```

```
70:❌            GasLimit:    k.computeGasLimit(sdkCtx),
```

computeGasLimit() returns the transaction's remaining gas, calculated via the gas meter. For simulations, it even returns a fixed value of `k.config.ContractSimulationGasLimit` (i.e., DefaultContractSimulationGasLimit = uint64(3_000_000)).

```
35: func (k Keeper) computeGasLimit(sdkCtx sdk.Context) uint64 {
36:     gasLimit := sdkCtx.GasMeter().Limit() -
sdkCtx.GasMeter().GasConsumedToLimit()
37:     if sdkCtx.ExecMode() == sdk.ExecModeSimulate {
38:             gasLimit = k.config.ContractSimulationGasLimit
39:     }
40:
41:     return gasLimit
42: }
```

The `GASLIMIT` EVM opcode uses this `GasLimit` value to return the block gas limit:

core/vm/instructions.go#L501-L504

```
func opGasLimit(pc *uint64, interpreter *EVMInterpreter, scope
*ScopeContext) ([]byte, error) {

scope.Stack.push(new(uint256.Int).SetUint64(interpreter.evm.Context.GasLi
mit))
        return nil, nil
}
```

Recommended mitigation steps

Consider setting `GasLimit` to the block gas limit instead of the transaction's gas.

leojin (Initia) confirmed and commented:

We fixed in this PR.

Initia mitigated:

Useds block gas limit in block context.

Status: Mitigation confirmed. Full details in reports from 0xAlix2, berndartmueller and Franfran.

[M-05] Amino legacy signing method broken because of name mismatch

Submitted by Franfran

The LegacyAmino codec is used for legacy reasons. Even if this codec is in the process of being deprecated for gogoprotobuf, it's still relevant and if not implemented correctly, it won't work correctly. Here is the implementation of the `RegisterConcrete` method to register the amino methods:

```
// This function should be used to register concrete types that will
appear in
// interface fields/elements to be encoded/decoded by go-amino.
// Usage:
// `amino.RegisterConcrete(MyStruct1{}, "com.tendermint/MyStruct1", nil)`
func (cdc *Codec) RegisterConcrete(o interface{}, name string, copts
*ConcreteOptions) {
```

As explained in the docs comment, it should be used to register concrete types for the interface. The `name` should match the `amino.name` options in the protobuf messages or it won't be able to find the interface and won't be able to encode/decode messages. Here are the places where the name is mismatched.

The `RegisterAminoMsg` can also be called which adds a check on the length of the input. However, it has not been found that any name exceeds this length of 39 characters. As a recommendation, the `RegisterAminoMsg` is preferred.

MiniEVM

- Missing `option (amino.name) = "evm/MsgUpdateParams"`:
  - Proto: https://github.com/initia-labs/minievm/blob/f5f60c398cd2316e99dfe42e7be09b8a05732c42/proto/minievm/evm/v1/tx.proto#L152
  - Go: https://github.com/initia-labs/minievm/blob/15346539ec6fec27b12812fd57b6ef1a1b01b12b/x/evm/types/codec.go#L18

MiniWASM

- `option (amino.name) = "move/MsgUpdateParams"`, should change to `"tokenfactory/MsgUpdateParams"`:
  - Proto: https://github.com/initia-labs/miniwasm/blob/01a654390846f4b6b8fe7a3febe05d203675e698/proto/miniwasm/tokenfactory/v1/tx.proto#L164
  - Go: https://github.com/initia-labs/miniwasm/blob/01a654390846f4b6b8fe7a3febe05d203675e698/x/tokenfactory/types/codec.go#L20

Initia

- `option (amino.name) = "move/MsgWhitelist"`, should change to `"move/MsgDelist"`:
  - Proto: https://github.com/initia-labs/initia/blob/6eea5ab6145063d897d802f5392a3fa4a3aaf01c/proto/initia/move/v1/tx.proto#L322
  - Go: https://github.com/initia-labs/initia/blob/53c7a45e5b492a265ded79d07c92aa4ac44607da/x/move/types/codec.go#L26
- `cdc.RegisterConcrete(Params{}, "ibchooks/Params", nil)`, should change to `"ibc-hooks/Params"` (and possibly rename the directory `ibchooks` of the proto files to `ibc-hooks`):
  - Proto: https://github.com/initia-labs/initia/blob/6eea5ab6145063d897d802f5392a3fa4a3aaf01c/proto/initia/ibchooks/v1/types.proto#L13
  - Go: https://github.com/initia-labs/initia/blob/def12e5be82de209a5bf2c30d1ddc6d94e6e9378/x/ibc-hooks/types/codec.go#L16
- `option (amino.name) = "mstake/StakeAuthorization"`, should change to `"mstaking/StakeAuthorization"`:
  - Proto: https://github.com/initia-labs/initia/blob/10ff76b8394c901e3f5d41350aa9822244c1030b/proto/initia/mstaking/v1/authz.proto#L14
  - Go: https://github.com/initia-labs/initia/blob/5de38c6fa15009889a413912eb69a280b2af9606/x/mstaking/types/codec.go#L26
- `Validators allow_list = 2 [(amino.oneof_name) = "mstake/StakeAuthorization/AllowList"]`, should change to `"mstaking/StakeAuthorization/AllowList"`:
  - Proto: https://github.com/initia-labs/initia/blob/10ff76b8394c901e3f5d41350aa9822244c1030b/proto/initia/mstaking/v1/authz.proto#L27
  - Go: https://github.com/initia-labs/initia/blob/5de38c6fa15009889a413912eb69a280b2af9606/x/mstaking/types/codec.go#L24
- `Validators deny_list = 3 [(amino.oneof_name) = "mstake/StakeAuthorization/DenyList"]`, should change to `"mstaking/StakeAuthorization/DenyList"`:
  - Proto: https://github.com/initia-labs/initia/blob/10ff76b8394c901e3f5d41350aa9822244c1030b/proto/initia/m

staking/v1/authz.proto#L29
    - Go: https://github.com/initia-labs/initia/blob/5de38c6fa15009889a413912eb69a280b2af9606/x/mstaking/types/codec.go#L25

Recommended mitigation steps

Make sure that the amino names are matching.

hoon (Initia) confirmed and commented:

Fixed in the commits Initia, MiniWasm and MiniEVM.

Initia mitigated:

Fixed amino.

Status: Mitigation confirmed. Full details in reports from berndartmueller, 0xAlix2 and Franfran.

[M-06] `MsgCreate2` deviates from EVM spec causing a large range of address not reachable

Submitted by oakcobalt

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/client/cli/tx.go#L89

Proof of Concept

Based on EIP-1014, salt used in `create2` is always 32 bytes.

The issue is `MsgCreate2` defines salt as `uin64`. This means a large range of address is not reachable in any flow that uses `MsgCreate2`.

```
// MsgCreate2 is a message to create a contract with the CREATE2 opcode.
type MsgCreate2 struct {
        // Sender is the that actor that signed the messages
        Sender string `protobuf:"bytes,1,opt,name=sender,proto3"
json:"sender,omitempty"`
        // Code is hex encoded raw contract bytes code.
        Code string `protobuf:"bytes,2,opt,name=code,proto3"
json:"code,omitempty"`
        // Salt is a random value to distinguish contract creation.
|>      Salt uint64 `protobuf:"varint,3,opt,name=salt,proto3"
json:"salt,omitempty"`
```

```
        // Value is the amount of fee denom token to transfer to the
contract.
        Value cosmossdk_io_math.Int
`protobuf:"bytes,4,opt,name=value,proto3,customtype=cosmossdk.io/math.Int
" json:"value"`
        // AccessList is a predefined list of Ethereum addresses and
their corresponding storage slots that a transaction will interact with
during its execution. can be none
        AccessList []AccessTuple
`protobuf:"bytes,5,rep,name=access_list,json=accessList,proto3"
json:"access_list"`
}
```

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/types/tx.pb.go#L171

In minievm, `MsgCreate2` is currently used as `create2` input in `CLI` and `gRPC` flows.

1. `create2` from `CLI`:

```
//x/evm/client/cli/tx.go
func Create2Cmd(ac address.Codec) *cobra.Command {
...
|>                      salt, err := strconv.ParseUint(args[0], 10, 64)
//@audit salt is parsed as uint64
                        if err != nil {
                                return errors.Wrap(err, "failed to parse
salt")
                        }
```

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/x/evm/client/cli/tx.go#L115

2. `create2` from `gRPC`:

```
func (c *msgClient) Create2(ctx context.Context, in *MsgCreate2, opts
...grpc.CallOption) (*MsgCreate2Response, error) {
        out := new(MsgCreate2Response)
        err := c.cc.Invoke(ctx, Msg_Create2_FullMethodName, in, out,
opts...)
        if err != nil {
                return nil, err
        }
```

```
        return out, nil
}
```

https://github.com/initia-labs/minievm/blob/744563dc6a642f054b4543db008df22664e4c125/api/minievm/evm/v1/tx_grpc.pb.go#L60

Impact

1. A large range of address not reachable through `CLI` and `gRPC`. This prevents certain `CLI`/`gRPC` based contracts deployments that would have been valid. This causes predictable failures for legitimate use cases.
2. Compatibility issue with other EVM based tooling.

Recommended mitigation steps

Consider allowing to use string for salt input.

leojin (Initia) disputed and commented:

There doesn't seem to be any other issues, just support for a smaller range of salts.

LSDan (judge) decreased severity to Medium and commented:

This is at root a compatibility issue that will impact the expected functionality of the chain. I don't think as reported it qualifies for high risk, but it does strike me as a valid medium.

beer-1 (Initia) commented:

Fixed here.

Initia mitigated:

Added salt range check.

Status: Unmitigated. Full details in reports from 0xAlix2, berndartmueller and Franfran, and also included in the Mitigation Review section below.

[M-07] Pool fraction is not truncated when allocating the tokens allowing to receive more rewards than owed

Submitted by ABAIKUNANBAEV

At the moment, the function `AllocateTokens()` uses the `Quo()` operation when calculating the pool fraction for a particular validator pool. The problem is that it does not round down to `0`, allowing to receive a bigger fraction than it actually is.

Proof of Concept

Let's take a look at how the operation is implemented:

https://github.com/initia-labs/initia/blob/main/x/distribution/keeper/allocation.go#L83

```
        poolFraction := rewardWeight.Weight.Quo(weightsSum)
```

Here, the `rewardWeight` is divided by the `WeightsSum` using the `Quo()` arithmetic and not `QuoTruncate()` that rounds the number down. The number is not truncated here as the `Quo()` function is used. The problem lies in the fact of how the `Quo()` method rounds the number:

`Quo()` calls `chopPrecisionAndRound()`:

https://github.com/piplabs/cosmos-sdk/blob/7ce4a34e92b12fc3aed8eec6e080b6493554072d/math/dec.go#L356

```
func (d LegacyDec) QuoMut(d2 LegacyDec) LegacyDec {
        // multiply by precision twice
        d.i.Mul(d.i, squaredPrecisionReuse)
        d.i.Quo(d.i, d2.i)

        chopPrecisionAndRound(d.i)
        if d.i.BitLen() > maxDecBitLen {
                panic("Int overflow")
        }
        return d
}
```

`QuoTruncate()` (which should be used instead) calls `chopPrecisionAndTruncate()` under the hood:

https://github.com/piplabs/cosmos-sdk/blob/7ce4a34e92b12fc3aed8eec6e080b6493554072d/math/dec.go#L376

```
// mutable quotient truncate
func (d LegacyDec) QuoTruncateMut(d2 LegacyDec) LegacyDec {
        // multiply precision twice
        d.i.Mul(d.i, squaredPrecisionReuse)
        d.i.Quo(d.i, d2.i)

        chopPrecisionAndTruncate(d.i)
        if d.i.BitLen() > maxDecBitLen {
                panic("Int overflow")
        }
```

```
        return d
}
```

The method always rounds down and has to be used instead; otherwise, the users will get the bigger `poolFraction` and therefore, bigger rewards. Take a look at the vanilla CosmosSDK implementation and how it rounds down the fractions:

https://github.com/cosmos/cosmos-sdk/blob/main/x/distribution/keeper/allocation.go#L71

```
powerFraction :=
math.LegacyNewDec(vote.Validator.Power).QuoTruncate(math.LegacyNewDec(tot
alPreviousPower))
```

Recommended mitigation steps

Use `QuoTruncate()` instead of `Quo()`.

beer-1 (Initia) confirmed and commented:

Seems to be a valid report. Fixed here.

Initia mitigated:

Use `quo truncate` at distribution pool fraction calculation.

Status: Mitigation confirmed. Full details in reports from 0xAlix2, berndartmueller and Franfran.

[M-08] IBC channel version negotiation bypass in IBC hooks middleware

Submitted by Rhaydden

https://github.com/initia-labs/initia//blob/f5cd97d3aadd694d511c61ce5f55a772f4d2b904/x/ibc-hooks/ibc_middleware.go#L61

Finding description and impact

The `OnChanOpenInit` function in `ibc_middleware.go` incorrectly returns the original version parameter instead of the negotiated `finalVersion` returned by the underlying app.

```
38: func (im IBCMiddleware) OnChanOpenInit(
39:     ctx sdk.Context,
40:     order channeltypes.Order,
```

```
41:     connectionHops []string,
42:     portID string,
43:     channelID string,
44:     channelCap *capabilitytypes.Capability,
45:     counterparty channeltypes.Counterparty,
46:     version string,
47: ) (string, error) {
48:     if hook, ok := im.ICS4Middleware.Hooks.
(OnChanOpenInitOverrideHooks); ok {
49:             return hook.OnChanOpenInitOverride(im, ctx, order,
connectionHops, portID, channelID, channelCap, counterparty, version)
50:     }
51:
52:     if hook, ok := im.ICS4Middleware.Hooks.
(OnChanOpenInitBeforeHooks); ok {
53:             hook.OnChanOpenInitBeforeHook(ctx, order, connectionHops,
portID, channelID, channelCap, counterparty, version)
54:     }
55:
56:     finalVersion, err := im.App.OnChanOpenInit(ctx, order,
connectionHops, portID, channelID, channelCap, counterparty, version)
57:
58:     if hook, ok := im.ICS4Middleware.Hooks.
(OnChanOpenInitAfterHooks); ok {
59:             hook.OnChanOpenInitAfterHook(ctx, order, connectionHops,
portID, channelID, channelCap, counterparty, version, finalVersion, err)
60:     }
61:     return version, err      ❌
62: }
```

The middleware ignores any version modifications made by the underlying application during channel initialization leading to version mismatches between channel endpoints. This renders `OnChanOpenInitOverrideHooks` completely ineffective for version negotiation.

```
16: type OnChanOpenInitOverrideHooks interface {
17:     OnChanOpenInitOverride(im IBCMiddleware, ctx sdk.Context, order
channeltypes.Order, connectionHops []string, portID string, channelID
string, channelCap *capabilitytypes.Capability, counterparty
channeltypes.Counterparty, version string) (string, error)
18: }
```

Also, it provides incorrect version information to `OnChanOpenInitAfterHooks`.

```
22: type OnChanOpenInitAfterHooks interface {
23:     OnChanOpenInitAfterHook(ctx sdk.Context, order
channeltypes.Order, connectionHops []string, portID string, channelID
string, channelCap *capabilitytypes.Capability, counterparty
channeltypes.Counterparty, version string, finalVersion string, err
error)
24: }
```

As a result, the version negotiation chain that hooks are designed to participate in broken.

Recommended mitigation steps

```
func (im IBCMiddleware) OnChanOpenInit(
    ctx sdk.Context,
    order channeltypes.Order,
    connectionHops []string,
    portID string,
    channelID string,
    channelCap *capabilitytypes.Capability,
    counterparty channeltypes.Counterparty,
    version string,
) (string, error) {
    if hook, ok := im.ICS4Middleware.Hooks.(OnChanOpenInitOverrideHooks);
ok {
        return hook.OnChanOpenInitOverride(im, ctx, order,
connectionHops, portID, channelID, channelCap, counterparty, version)
    }

    if hook, ok := im.ICS4Middleware.Hooks.(OnChanOpenInitBeforeHooks);
ok {
        hook.OnChanOpenInitBeforeHook(ctx, order, connectionHops, portID,
channelID, channelCap, counterparty, version)
    }

    finalVersion, err := im.App.OnChanOpenInit(ctx, order,
connectionHops, portID, channelID, channelCap, counterparty, version)

    if hook, ok := im.ICS4Middleware.Hooks.(OnChanOpenInitAfterHooks); ok
{
        hook.OnChanOpenInitAfterHook(ctx, order, connectionHops, portID,
channelID, channelCap, counterparty, version, finalVersion, err)
    }
-   return version, err
+   return finalVersion, err
```

```
}
```

beer-1 (Initia) confirmed and commented:

Fixed here.

Initia mitigated:

Fixed to return final version.

Status: Mitigation confirmed. Full details in reports from 0xAlix2, berndartmueller and Franfran.

Low Risk and Non-Critical Issues

For this audit, 7 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Rhaydden** received the top score from the judge.

The following wardens also submitted reports: ABAIKUNANBAEV, Franfran, gh8eo, gxh191, oakcobalt, and Sparrow.

Note: C4 excluded the invalid entries determined by the judge from this report.

[01] Incorrect module name in telemetry measurement for bank module's `InitGenesis`

In the bank module's InitGenesis method, the telemetry measurement is incorrectly using "crisis" as the module name instead of "bank". This means the initialization time is being attributed to the wrong module.

Makes it difficult to accurately track the performance of the bank module's genesis initialization.

https://github.com/initia-labs/minievm/blob/0bc0c0f19de493c3c711f6f3d48e5befb0c48115/x/bank/module.go#L125

```
// in x/bank/module.go
func (am AppModule) InitGenesis(ctx sdk.Context, cdc codec.JSONCodec,
data json.RawMessage) {
    start := time.Now()
    var genesisState types.GenesisState
    cdc.MustUnmarshalJSON(data, &genesisState)
    telemetry.MeasureSince(start, "InitGenesis", "crisis", "unmarshal")
// incorrect module name

    am.keeper.InitGenesis(ctx, &genesisState)
```

```
}
```

Fix

Update the telemetry measurement to use the correct module name:

```
func (am AppModule) InitGenesis(ctx sdk.Context, cdc codec.JSONCodec,
data json.Ra
     start := time.Now()
     var genesisState types.GenesisState
     cdc.MustUnmarshalJSON(data, &genesisState)
-    telemetry.MeasureSince(start, "InitGenesis", "crisis", "unmarshal")
+    telemetry.MeasureSince(start, "InitGenesis", "bank", "unmarshal")

     am.keeper.InitGenesis(ctx, &genesisState)
 }
```

[02] Incorrect example command in `GetCmdQueryRedelegations help` text

The `GetCmdQueryRedelegations` function shows an incorrect example command in its `help` text. The command is defined as "redelegations" (plural) in the `Use` field:

https://github.com/initia-labs/initia//blob/f5cd97d3aadd694d511c61ce5f55a772f4d2b904/x/mstaking/client/cli/query.go#L588-L592

```
Use:   "redelegations [delegator-addr]",
```

Albeit, the example in the `help` text uses "redelegation" (singular):

```
fmt.Sprintf(`Query all redelegation records for an individual delegator.

Example:
$ %s query staking redelegation %s1gghjut3ccd8ay0zduzj64hwre2fxs9ld75ru9p
`,
    version.AppName, bech32PrefixAccAddr,
),
```

This mismatch would cause the example command to fail if users try to copy and use it.

Fix

Update the example in the `help` text to use the correct plural form:

```
fmt.Sprintf(`Query all redelegation records for an individual delegator.

Example:
+  $ %s query staking redelegations
%s1gghjut3ccd8ay0zduzj64hwre2fxs9ld75ru9p
-  $ %s query staking redelegation
%s1gghjut3ccd8ay0zduzj64hwre2fxs9ld75ru9p
`,
    version.AppName, bech32PrefixAccAddr,
),
```

[03] Incorrect telemetry label in `ScriptJSON` method

The `ScriptJSON` method uses the telemetry label "script", which is the same label used by the Script method. This causes both `Script` and `ScriptJSON` executions to be logged under the same metric, making it impossible to distinguish between them.

https://github.com/initia-labs/initia//blob/f5cd97d3aadd694d511c61ce5f55a772f4d2b904/x/move/keeper/msg_server.go#L179-L180

```
func (ms MsgServer) ScriptJSON(context context.Context, req
*types.MsgScriptJSON) (*types.MsgScriptJSONResponse, error) {
    defer telemetry.MeasureSince(time.Now(), "move", "msg", "script")
```

Fix

```
 func (ms MsgServer) ScriptJSON(context context.Context, req
*types.MsgScriptJSON) (*types.MsgScriptJSONResponse, error) {
-    defer telemetry.MeasureSince(time.Now(), "move", "msg", "script")
+    defer telemetry.MeasureSince(time.Now(), "move", "msg",
"script_json")
```

[04] `StableSwap` whitelist function allows non-existent pools to be whitelisted

`Whitelist()` function in the StableSwap keeper returns success (`nil` error) for non-existent pools. This goes against the function's intended purpose of validating pools for whitelisting.

When `HasPool` returns false, indicating the pool doesn't exist, the function returns `nil` instead of an error:

https://github.com/initia-labs/initia//blob/f5cd97d3aadd694d511c61ce5f55a772f4d2b904/x/move/keeper/stableswap.go#L54-L56

```
if !ok {
    return nil  // Incorrectly indicates successful whitelisting
}
```

This allows non-existent pools to be whitelisted. Attackers could pre-whitelist addresses and later create pools with malicious configurations. This bypasses the entire whitelist check.

Add this test:

```
func Test_StableSwap_Whitelist_NonExistentPool(t *testing.T) {
        ctx, input := createDefaultTestInput(t)

        stableSwapKeeper := keeper.NewStableSwapKeeper(&input.MoveKeeper)

        // Create a random non-existent pool address
        nonExistentPool := vmtypes.AccountAddress{1, 2, 3, 4, 5, 6, 7, 8,
9, 10, 11, 12, 13, 14, 15, 16}

        // This should return an error for a non-existent pool, but due
to the bug it returns nil
        err := stableSwapKeeper.Whitelist(ctx, nonExistentPool)
        require.NoError(t, err) // This will pass due to the bug!

        // Verify the pool doesn't actually exist
        hasPool, err := stableSwapKeeper.HasPool(ctx, nonExistentPool)
        require.NoError(t, err)
        require.False(t, hasPool, "Pool should not exist but was
successfully whitelisted!")
}
```

Logs

```
Running tool: /usr/local/go/bin/go test -timeout 30s -run
^Test_StableSwap_Whitelist_NonExistentPool$ github.com/initia-
labs/initia/x/move/keeper
```

```
=== RUN   Test_StableSwap_Whitelist_NonExistentPool
--- PASS: Test_StableSwap_Whitelist_NonExistentPool (0.32s)
PASS
ok      github.com/initia-labs/initia/x/move/keeper     12.851s
```

Fix

The `whitelist()` function should return an error when the pool doesn't exist.

[05] Interface implementation check for `ShorthandAccount` is duplicated but the check for `ContractAccount` is missing

`auth.go` has a duplicate interface compliance check that masks a missing check for the `ContractAccount` type.

https://github.com/initia-labs/minievm/blob/0bc0c0f19de493c3c711f6f3d48e5befb0c48115/x/evm/types/auth.go#L13-L22

```
var (
    _ sdk.AccountI = (*ShorthandAccount)(nil)  // Appears twice
    _ sdk.AccountI = (*ShorthandAccount)(nil)  // Duplicate line

    _ authtypes.GenesisAccount = (*ContractAccount)(nil)
    _ authtypes.GenesisAccount = (*ShorthandAccount)(nil)

    _ ShorthandAccountI = (*ShorthandAccount)(nil)
)
```

While this doesn't cause runtime issues, it means we're missing compile-time verification that `ContractAccount` implements `sdk.AccountI`. This could allow interface implementation bugs to slip through if the `ContractAccount` type is modified in the future.

Fix

Replace the duplicate check with the missing `ContractAccount` verification:

```
 var (
-    _ sdk.AccountI = (*ShorthandAccount)(nil)
+    _ sdk.AccountI = (*ContractAccount)(nil)
    _ sdk.AccountI = (*ShorthandAccount)(nil)

     _ authtypes.GenesisAccount = (*ContractAccount)(nil)
```

```
    _ authtypes.GenesisAccount = (*ShorthandAccount)(nil)
```

[06] Redundant coin validation in send function creates unnecessary gas overhead

In the `Send` function, there are two consecutive validations for coin amounts that perform overlapping checks:

https://github.com/initia-labs/minievm/blob/0bc0c0f19de493c3c711f6f3d48e5befb0c48115/x/bank/keeper/msg_server.go#L50-L56

```
// Current implementation
if !msg.Amount.IsValid() {
    return nil, errorsmod.Wrap(sdkerrors.ErrInvalidCoins,
msg.Amount.String())
}

if !msg.Amount.IsAllPositive() {
    return nil, errorsmod.Wrap(sdkerrors.ErrInvalidCoins,
msg.Amount.String())
}
```

The `IsValid()` method already includes the positive amount check that `IsAllPositive()` performs, making the second check redundant. This creates unnecessary gas consumption for users sending transactions.

Also, the current implementation allows empty transfers (`msg.Amount` with length 0) to pass validation.

Fix

Replace the redundant validation with a single, comprehensive check. Something like:

```
// Check for empty transfers first
if len(msg.Amount) == 0 {
    return nil, errorsmod.Wrap(sdkerrors.ErrInvalidCoins, "empty amount")
}

// Single validation check
if !msg.Amount.IsValid() {
    return nil, errorsmod.Wrap(sdkerrors.ErrInvalidCoins,
msg.Amount.String())
```

```
}
```

[07] Attacker could exhaust resources because `AllBalances` has unbounded pagination in bank module

Note from Sponsor: "cosmos limits that to 100, and normally we are setting reverse proxy to limit that".

Finding Description and Impact

The Bank module's `AllBalances` GRPC query endpoint doesn't have pagination limits which allows clients to request arbitrarily large result sets. An attacker could exploit this to cause significant resource consumption and DoS the blockchain nodes.

In grpc_query.go, `AllBalances` method accepts pagination params without validation:

```
balances, pageRes, err := k.mk.GetPaginatedBalances(ctx, req.Pagination,
addr)
```

An attacker can force nodes to:

1. Spend significant CPU time processing large result sets
2. Cause a DoS by increasing response latency for other queries. Some nodes will crash under extreme load.

Proof of concept

Included this fuzztest in QA because I couldn't find the 100 limit implemented by Cosmos. I added this test to `grpc_query_test.go` to request 100,000 tokens with a 1 billion page size. Ran test with `go test -v ./x/bank/keeper -run TestQueryAllBalancesWithExtremeLoad`.

Add these to the imports:

```
        "fmt"
        "time"
        "runtime"
```

```
func TestQueryAllBalancesWithExtremeLoad(t *testing.T) {
        ctx, input := createDefaultTestInput(t)
        _, _, addr := testdata.KeyTestPubAddr()
```

```
        // Create an account with an extremely large number of different
token balances
        acc := input.AccountKeeper.NewAccountWithAddress(ctx, addr)
        input.AccountKeeper.SetAccount(ctx, acc)

        // Create 100,000 different token balances with large amounts
        totalCoins := sdk.NewCoins()
        for i := 0; i < 100000; i++ {
                // Use large amounts to increase memory usage
                amount := math.NewInt(1000000000000) // 1 trillion units
per token
                coin := sdk.NewCoin(fmt.Sprintf("token%d", i), amount)
                totalCoins = totalCoins.Add(coin)
        }
        input.Faucet.Fund(ctx, acc.GetAddress(), totalCoins...)

        // Test with extremely large page size
        pageReq := &query.PageRequest{
                Key:        nil,
                Limit:      1000000000, // 1 billion page size
                CountTotal: true,
        }
        req := types.NewQueryAllBalancesRequest(addr, pageReq, false)

        // Record memory stats before query
        var m runtime.MemStats
        runtime.ReadMemStats(&m)
        beforeAlloc := m.Alloc
        beforeSys := m.Sys

        // Record start time
        startTime := time.Now()

        // Execute query
        res, err := input.BankKeeper.AllBalances(ctx, req)

        // Calculate duration
        duration := time.Since(startTime)

        // Record memory stats after query
        runtime.ReadMemStats(&m)
        afterAlloc := m.Alloc
        afterSys := m.Sys

        // Log performance metrics
        t.Logf("Query execution time: %v", duration)
```

```
        t.Logf("Memory allocated before: %d MB", beforeAlloc/1024/1024)
        t.Logf("Memory allocated after: %d MB", afterAlloc/1024/1024)
        t.Logf("Memory increase: %d MB", (afterAlloc-
beforeAlloc)/1024/1024)
        t.Logf("System memory before: %d MB", beforeSys/1024/1024)
        t.Logf("System memory after: %d MB", afterSys/1024/1024)
        t.Logf("System memory increase: %d MB", (afterSys-
beforeSys)/1024/1024)

        // Test should still complete successfully despite the load
        require.NoError(t, err)
        require.NotNil(t, res)
        require.Equal(t, 100000, len(res.Balances))

        // Verify total amount (sample a few random coins)
        for i := 0; i < 10; i++ {
                randomIndex := i * 10000 // Check every 10000th coin
                denom := fmt.Sprintf("token%d", randomIndex)
                require.True(t,
res.Balances.AmountOf(denom).Equal(math.NewInt(1000000000000)))
        }
}
```

It panics because the test timed out after about 10 minutes:

```
=== RUN   TestQueryAllBalancesWithExtremeLoad
panic: test timed out after 10m0s
        running tests:
                TestQueryAllBalancesWithExtremeLoad (10m0s)

goroutine 52 [running]:
testing.(*M).startAlarm.func1()
        /Users/rhaydden/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.23.6.darwin-amd64/src/testing/testing.go:2373 +0x385
created by time.goFunc
        /Users/rhaydden/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.23.6.darwin-amd64/src/time/sleep.go:215 +0x2d

goroutine 1 [chan receive, 10 minutes]:
testing.(*T).Run(0xc00159a000, {0x1049a936a?, 0x300745d801307b50?},
0x105a37388)
        /Users/rhaydden/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.23.6.darwin-amd64/src/testing/testing.go:1751 +0x3ab
testing.runTests.func1(0xc00159a000)
```

```
        /Users/rhaydden/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.23.6.darwin-amd64/src/testing/testing.go:2168 +0x37
testing.tRunner(0xc00159a000, 0xc001307c70)
        /Users/rhaydden/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.23.6.darwin-amd64/src/testing/testing.go:1690 +0xf4
testing.runTests(0xc000013560, {0x1077e34c0, 0x1c, 0x1c}, {0x102252250?,
0x0?, 0x1078ee400?})
        /Users/rhaydden/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.23.6.darwin-amd64/src/testing/testing.go:2166 +0x43d
testing.(*M).Run(0xc000e1f180)
        /Users/rhaydden/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.23.6.darwin-amd64/src/testing/testing.go:2034 +0x64a
main.main()
        _testmain.go:101 +0x9b

goroutine 20 [select, 10 minutes]:
github.com/desertbit/timer.timerRoutine()
        /Users/rhaydden/go/pkg/mod/github.com/desertbit/timer@v0.0.0-
20180107155436-c41aec40b27f/timers.go:119 +0x9e
created by github.com/desertbit/timer.init.0 in goroutine 1
        /Users/rhaydden/go/pkg/mod/github.com/desertbit/timer@v0.0.0-
20180107155436-c41aec40b27f/timers.go:15 +0x1a

goroutine 37 [select]:
go.opencensus.io/stats/view.(*worker).start(0xc000679e00)

/Users/rhaydden/go/pkg/mod/go.opencensus.io@v0.24.0/stats/view/worker.go:
292 +0x9f
created by go.opencensus.io/stats/view.init.0 in goroutine 1

/Users/rhaydden/go/pkg/mod/go.opencensus.io@v0.24.0/stats/view/worker.go:
34 +0x8d

goroutine 66 [runnable]:
math/big.(*Int).Add(0xc001e52800, 0xc001e527c0?, 0xc001e238a0?)
        /Users/rhaydden/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.23.6.darwin-amd64/src/math/big/int.go:141 +0x1c5
cosmossdk.io/math.add(...)
        /Users/rhaydden/go/pkg/mod/cosmossdk.io/math@v1.4.0/int.go:46
cosmossdk.io/math.Int.SafeAdd({0xc001262000?}, {0x11296e108?})
        /Users/rhaydden/go/pkg/mod/cosmossdk.io/math@v1.4.0/int.go:293
+0x34
cosmossdk.io/math.Int.Add(...)
        /Users/rhaydden/go/pkg/mod/cosmossdk.io/math@v1.4.0/int.go:279
github.com/cosmos/cosmos-sdk/types.Coin.Add({{0xc002228940?,
0xc002309a78?}, {0xc001e527c0?}}, {{0xc002228940?, 0x10598da00?},
```

```
{0xc001e238a0?}})
        /Users/rhaydden/go/pkg/mod/github.com/cosmos/cosmos-
sdk@v0.50.11/types/coin.go:114 +0x85
github.com/cosmos/cosmos-sdk/types.Coins.safeAdd({0xc002f7e000, 0x9eea,
0xb2aa}, {0xc00230d1b8, 0x1, 0x1})
        /Users/rhaydden/go/pkg/mod/github.com/cosmos/cosmos-
sdk@v0.50.11/types/coin.go:344 +0x4c9
github.com/cosmos/cosmos-sdk/types.Coins.Add(...)
        /Users/rhaydden/go/pkg/mod/github.com/cosmos/cosmos-
sdk@v0.50.11/types/coin.go:314
github.com/initia-
labs/initia/x/bank/keeper_test.TestQueryAllBalancesWithExtremeLoad(0xc001
59a1a0)
        /Users/rhaydden/initia-2/x/bank/keeper/grpc_query_test.go:122
+0x437
testing.tRunner(0xc00159a1a0, 0x105a37388)
        /Users/rhaydden/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.23.6.darwin-amd64/src/testing/testing.go:1690 +0xf4
created by testing.(*T).Run in goroutine 1
        /Users/rhaydden/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.23.6.darwin-amd64/src/testing/testing.go:1743 +0x390
FAIL    github.com/initia-labs/initia/x/bank/keeper     606.984s
FAIL
```

I also made a POC that shows significant memory growth even with a moderate number of tokens (10,000 tokens):

▶ Details

It outputs:

```
=== RUN   TestQueryAllBalancesWithExtremeLoad
    grpc_query_test.go:119: Creating 10000 token balances...
    grpc_query_test.go:140: Created 0 tokens...
    grpc_query_test.go:140: Created 1000 tokens...
    grpc_query_test.go:140: Created 2000 tokens...
    grpc_query_test.go:140: Created 3000 tokens...
    grpc_query_test.go:140: Created 4000 tokens...
    grpc_query_test.go:140: Created 5000 tokens...
    grpc_query_test.go:140: Created 6000 tokens...
    grpc_query_test.go:140: Created 7000 tokens...
    grpc_query_test.go:140: Created 8000 tokens...
    grpc_query_test.go:140: Created 9000 tokens...
    grpc_query_test.go:144: Finished creating tokens. Starting query
test...
    grpc_query_test.go:175: Query execution time: 120.322792ms
```

```
    grpc_query_test.go:176: Memory allocated before: 228 MB
    grpc_query_test.go:177: Memory allocated after: 246 MB
    grpc_query_test.go:178: Memory increase: 17 MB
    grpc_query_test.go:179: System memory before: 344 MB
    grpc_query_test.go:180: System memory after: 344 MB
    grpc_query_test.go:181: System memory increase: 0 MB
--- PASS: TestQueryAllBalancesWithExtremeLoad (50.35s)
PASS
```

We can observe initial memory allocation of 303 MB and final memory allocation of 321 MB. A memory increase of 17 MB.

System memory increase is 12 MB. This is just for a single query with 10,000 tokens. In a production environment with multiple concurrent requests, multiple users and larger token sets, the memory usage could easily become unmanageable.

Fix

Confirm if cosmos indeed limits that to 100. If not, implement a maximum page size limit maybe like 100 items per page in the `AllBalances` query endpoint.

[08] Fix error message in `ExecuteAuthorization:ValidateBasic`

The error message in `ExecuteAuthorization::ValidateBasic` method incorrectly states "invalid module names" when validating empty function names, which could mislead users debugging authorization issues. The error message should accurately reflect that the validation is checking function names, not module names.

https://github.com/initia-labs/initia//blob/f5cd97d3aadd694d511c61ce5f55a772f4d2b904/x/move/types/authz.go#L124

Fix

```
func (a ExecuteAuthorization) ValidateBasic() error {
            return errors.Wrapf(sdkerrors.ErrInvalidRequest, "invalid
module name: %s", v.ModuleName)
        }
        if len(v.FunctionNames) == 0 {
-           return errors.Wrap(sdkerrors.ErrInvalidRequest, "invalid
module names")
+           return errors.Wrap(sdkerrors.ErrInvalidRequest, "function
names cannot be empty")
        }
```

Mitigation Review

Introduction

Following the C4 audit, 4 wardens (berndartmueller, Franfran and a_kalout and ali_shehab of team 0xAlix2) reviewed the mitigations implemented by the Initia team. Additional details can be found within the C4 Initia Cosmos Mitigation Review repository.

Mitigation Review Scope

URL	MITIGATION OF	PURPOSE
Link	H-05	Intrinsic gas
Link	H-06	Introduce to use submsg gas limit
Link	H-03	Fix to maintain execute request on snapshot
Link	H-04	Add max addresses limit
Link	H-07	Change to return exact error only at simulate or checktx
Link	H-08	Change to return exact error only at simulate or checktx
Link	H-01	Burn coins error
Link	H-02	fix: `ConvertCosmosTxToEthereumTx` to properly check type url
Link	M-04	Use block gas limit in block context
Link 1, Link 2, Link 3	M-05	Fix amino
Link	M-06	Add salt range check
Link	M-01	Delete cosmwasm address validation
Link	M-08	Fix to return final version
Link	M-07	Use quo truncate at distribution pool fraction calculation

URL	MITIGATION OF	PURPOSE
Link	ADD-01	Apply Ottersec audit

Note: test files are out of scope

Out of Scope

- M-02: Contract deployment restriction can be bypassed
- M-03: `COINBASE` opcode returns an empty address instead of the block proposer resulting in incompatibility with the EVM

Mitigation Review Summary

The wardens confirmed the mitigations for all in-scope findings except for H-06, H-08 and M-06, which were all unmitigated. They also surfaced 5 new issues, all of Medium severity. The table below provides details regarding the status of each in-scope vulnerability from the original audit, followed by full details on the in-scope vulnerabilities that were unmitigated, as well as newly discovered issues.

ORIGINAL ISSUE	STATUS	FULL DETAILS
H-01	Mitigation Confirmed	Reports from berndartmueller, Franfran and 0xAlix2
H-02	Mitigation Confirmed	Reports from berndartmueller, Franfran and 0xAlix2
H-03	Mitigation Confirmed	Reports from 0xAlix2, berndartmueller and Franfran
H-04	Mitigation Confirmed	Reports from 0xAlix2, berndartmueller and Franfran
H-05	Mitigation Confirmed	Reports from 0xAlix2, berndartmueller and Franfran
H-06	Unmitigated	Reports from berndartmueller and 0xAlix2
H-07	Mitigation Confirmed	Reports from 0xAlix2, berndartmueller and Franfran

ORIGINAL ISSUE	STATUS	FULL DETAILS
H-08	Unmitigated	Reports from berndartmueller, 0xAlix2 and Franfran
M-01	Mitigation Confirmed	Reports from berndartmueller, 0xAlix2 and Franfran
M-04	Mitigation Confirmed	Reports from 0xAlix2, berndartmueller and Franfran
M-05	Mitigation Confirmed	Reports from berndartmueller, 0xAlix2 and Franfran
M-06	Unmitigated	Reports from 0xAlix2, berndartmueller and Franfran
M-07	Mitigation Confirmed	Reports from 0xAlix2, berndartmueller and Franfran
M-08	Mitigation Confirmed	Reports from 0xAlix2, berndartmueller and Franfran
ADD-01	Mitigation Confirmed	Reports from berndartmueller and 0xAlix2

[H-06] Unmitigated

Submitted by berndartmueller, also found by 0xAlix2

Original Issue

H-06: https://code4rena.com/audits/2025-02-initia-cosmos/submissions/F-11

Finding description and impact

In the new mitigation, a gas limit must be provided when calling `execute_cosmos` or `execute_cosmos_with_options` in Solidity. This gas limit is pre-charged in the precompile, which makes sure that there's sufficient gas.

During message dispatching, this gas limit is used in a local gas meter to restrict the message's gas.

However, this local gas meter (`ctx`) is only used by the main `handler` call. The callback EVM call is still using the `parentCtx` (see here) which is not using the local gas meter, rather the

parent gas meter.

Note that the same issue applies to the callback call in the defer block of the dispatchMessage() function. The callback EVM call also uses the same gas meter as the parent call.

This was mentioned in the original F-11 submission.

Links to affected code

context.go#L600

[H-08] Unmitigated

Submitted by berndartmueller, also found by 0xAlix2 and Franfran

Original Issue

H-08: https://code4rena.com/audits/2025-02-initia-cosmos/submissions/F-135

Finding description and impact

The issue is not fixed. The fix provided is for F-13.

`ctx.GasMeter().ConsumeGas(storetypes.Gas(len(input))*GAS_PER_BYTE, "input bytes")` is still called after:

- Calling `MethodById()`
- Calling `Unpack()`

Links to affected code

contract.go#L128-L139

beer-1 (Initia) commented:

Fixed here.

[M-06] Unmitigated

Submitted by 0xAlix2, also found by berndartmueller and Franfran

Original Issue

M-06: https://code4rena.com/audits/2025-02-initia-cosmos/submissions/F-26

Finding description and impact

The issue arises because `MsgCreate2` defines `salt` as a `uint64`, which is only 8 bytes, instead of the required **32 bytes** as per EIP-1014.

This restriction prevents a large range of addresses from being generated, making `CREATE2` deployments non-compliant with the EVM specification.

However, the issue remains unresolved.

Mitigation

Commit 3650a360d88896a29ddcdee6f2d9060847e6349b attempted to address this by validating that the input salt is `≤ MaxUint32`.

However, this is still incorrect:

- `MaxUint32` is only 4 bytes, while `CREATE2` requires 32 bytes.
- The issue persists, as a large range of addresses remain unreachable.

Recommended Mitigation Steps

The correct solution is to change the salt type to `[32]byte` to fully comply with `EIP-1014` and ensure that all valid `CREATE2` addresses are reachable.

Links to affected code

tx.pb.go#L171

beer-1 (Initia) commented via duplicate issue #5:

Misunderstood it as 32 bits. Fixed here.

Panic due to OOG during message dispatching can be misused

Submitted by berndartmueller

Severity: Medium

Finding description and impact

Compared to previously, `dispatchMessage()` now panics if the dispatched messages cause an out-of-gas error, supposedly to propagate the error.

While, in theory, this sounds like a good mitigation to prevent further execution when gas is fully consumed, it causes a new issue where a nested/child Solidity call can purposefully trigger the panic to revert the whole transaction (which is not necessarily always possible in Solidity, e.g., when using try/catch).

Proof of Concept

Consider a Solidity relayer contract that interacts and calls potentially "untrusted" child contracts via a try/catch and a gas limit. While the child contract call can fail, the overall call (transaction) must always succeed (similar to Layer Zero).

However, this child contract can cause the overall transaction to fail by dispatching a message that will run out of gas, and thus causing the panic in `dispatchMessage()`. This represents a DoS for the relayer contract, potentially locking funds.

Another example would be nested EVMCallWithTracer calls, called from within a dispatched `execute_cosmos` Solidity call. If it panics due to running out of gas in the nested gas meter, it would panic the overall transaction, even though the parent `EVMCallWithTracer` call has sufficient gas.

Please note that these are just two examples of how this panic might cause troubles.

Recommended mitigation steps

Do not `panic`.

Links to affected code

context.go#L556

beer-1 (Initia) commented

In this dispatch model, we cannot allow normal evm try catch.

We also know this problem, and agree with you this can break normal evm experience. I think we can introduce a new method to disable execute cosmos messages.

If this flag set, then the child calls cannot execute cosmos messages.

Fixed here, here and here.

`Value` is not provided to `QueryCallRequest` in `eth_call`, preventing reliable simulations

Submitted by berndartmueller

Severity: Medium

Finding description and impact

In PR #165, I noticed the change which adds the `AccessList` to `QueryCallRequest`, so that the `Call` simulation properly works.

See here.

However, it seems that `Value` is missing. This prevents reliably using `eth_call` to simulate a transaction that requires and expects a non-zero value to function.

Recommended mitigation steps

Consider adding `Value` to `QueryCallRequest`.

Links to affected code

eth.go#L89-L94

beer-1 (Initia) commented:

`value` was in the call request. Please check here.

berndartmueller (warden) commented:

As the sponsor correctly pointed out, `Value` is a property of `QueryCallRequest` in the protobuf definition, as seen here.

However, this does not mean that this submission is invalid.

As pointed out in the submission, `Value` is not provided to `QueryCallRequest`, e.g., when using the `eth_call` JSONRPC endpoint (e.g., can be used for simulations).

In other instances, for example, when using the `evm call` CLI command, `Value` is set.

Therefore, I kindly ask the judge to have another look at my submission and re-evaluate the validity.

LSDan (judge) commented:

Thanks for expanding on that. In reevaluating the whole function, I see that you are indeed correct. Validity is reinstated.

leojin (Initia) commented

Fixed here.

Minievm's JSON RPC doesn't have a limit on the user's queued transactions, allowing anyone to fill it up

Submitted by 0xAlix2

Severity: Medium

Finding description and impact

Minievm provides a JSON RPC that allows users to clients to interact with Ethereum nodes using JSON (JavaScript Object Notation) messages over HTTP or IPC (Inter-Process Communication), it provides multiple requests/calls.

`eth_sendRawTransaction` allows users to send raw Minievm transactions using that RPC. When an `eth_sendRawTransaction` call is sent, `SendRawTransaction` gets called, which does multiple checks and processes, like signature validation, we don't care about that here. What we care about here is that it grabs the TX sequence from the signature, and the account's sequence/nonce, compares it, and blocks `sequence > transaction sequence`.

https://github.com/initia-labs/minievm/blob/e960a514d4423d9ea19147b4dc1c8e5e9cc6d2a1/jsonrpc/backend/tx.go#L85-L87:

```
if accSeq > txSeq {
        return fmt.Errorf("%w: next nonce %v, tx nonce %v",
core.ErrNonceTooLow, accSeq, txSeq)
}
```

`If` doesn't enforce equality as it also holds queued transactions; these transactions are "un-executable" because of nonce mismatch. In other words, if my account's sequence is 5, if I submit an `eth_sendRawTransaction` call with `sequence >= 6`, it'll get queued rather than being executed right away.

These transactions are saved in an LRU cache, called `queuedTxs`. This LRU has a limit that is set when starting up the RPC.

https://github.com/initia-labs/minievm/blob/e960a514d4423d9ea19147b4dc1c8e5e9cc6d2a1/jsonrpc/backend/backend.go#L111:

```
queuedTxs, err := lrucache.NewWithEvict(cfg.QueuedTransactionCap, func(_
string, txCache txQueueItem) { ... }
```

However, this is a global limit, and there's no limit on the number of transactions that a user could queue. When it is full and new items come in, the first inserted item gets evicted. This allows any user to fill up the transactions queue repeatedly and block other users from queueing up transactions.

As a side note, Ethereum (Geth) sets a max queued transitions per account to 64.

Proof of Concept

Add the following test `minievm/jsonrpc/backend/tx_test.go`:

```
func generate_and_send_tx(
        t *testing.T,
```

```
        app *app.MinitiaApp,
        ctx sdk.Context,
        backend *backend.JSONRPCBackend,
        contractAddr []byte,
        user common.Address,
        pk *ecdsa.PrivateKey,
        nonce uint64) {
        tx, _ := tests.GenerateTransferERC20Tx(
                t, app, pk,
                common.BytesToAddress(contractAddr), user,
new(big.Int).SetUint64(1_000_000), tests.SetNonce(nonce))
        evmTx, _, err :=
evmkeeper.NewTxUtils(app.EVMKeeper).ConvertCosmosTxToEthereumTx(ctx, tx)
        require.NoError(t, err)
        require.NotNil(t, evmTx)

        txBz, err := evmTx.MarshalBinary()
        require.NoError(t, err)
        _, err = backend.SendRawTransaction(txBz)
        require.NoError(t, err)
}

func Test_JsonRPC_FillUpTxsQ(t *testing.T) {
        input := setupBackend(t)
        app, _, backend, addrs, privKeys := input.app, input.addrs,
input.backend, input.addrs, input.privKeys

        bob, bob_pk := addrs[0], privKeys[0]
        alice, alice_pk := addrs[1], privKeys[1]

        tx, _ := tests.GenerateCreateERC20Tx(t, app, bob_pk)
        _, finalizeRes := tests.ExecuteTxs(t, app, tx)
        tests.CheckTxResult(t, finalizeRes.TxResults[0], true)

        tx, _ = tests.GenerateCreateERC20Tx(t, app, alice_pk)
        _, finalizeRes = tests.ExecuteTxs(t, app, tx)
        tests.CheckTxResult(t, finalizeRes.TxResults[0], true)

        events := finalizeRes.TxResults[0].Events
        createEvent := events[len(events)-3]
        require.Equal(t, evmtypes.EventTypeContractCreated,
createEvent.GetType())

        contractAddr, err :=
hexutil.Decode(createEvent.Attributes[0].Value)
        require.NoError(t, err)
```

```
        ctx, err := app.CreateQueryContext(0, false)
        require.NoError(t, err)

        // Bob fills the transaction queue with 1_000 transactions
        for i := 10; i < 1_010; i++ {
                generate_and_send_tx(t, app, ctx, backend, contractAddr,
bob, bob_pk, uint64(i))
        }

        txPool, err := backend.TxPoolContent()
        require.NoError(t, err)

        // Bob has 1_000 transactions in the queue
        require.Equal(t, len(txPool["queued"][bob.String()]), 1_000)
        // Alice has 0 transactions in the queue
        require.Equal(t, len(txPool["queued"][alice.String()]), 0)

        // Alice queues 1 transaction
        generate_and_send_tx(t, app, ctx, backend, contractAddr, alice,
alice_pk, 100)

        txPool, err = backend.TxPoolContent()
        require.NoError(t, err)

        // Bob has 999 transactions in the queue
        require.Equal(t, len(txPool["queued"][bob.String()]), 999)
        // Alice has 1 transaction in the queue
        require.Equal(t, len(txPool["queued"][alice.String()]), 1)

        // Bob fills the transaction queue with 1_000 transactions again
        for i := 1_010; i < 2_010; i++ {
                generate_and_send_tx(t, app, ctx, backend, contractAddr,
bob, bob_pk, uint64(i))
        }

        txPool, err = backend.TxPoolContent()
        require.NoError(t, err)

        // Bob has 1_000 transactions in the queue
        require.Equal(t, len(txPool["queued"][bob.String()]), 1_000)
        // Alice has 1 transaction in the queue
        require.Equal(t, len(txPool["queued"][alice.String()]), 0)
}
```

Recommended mitigation steps

Instead of having a hard global limit on the queued transactions LRU cache, have a limit for each user's queued transactions, similar to what's done in Ethereum (Geth).

Links to affected code

tx.go#L95

beer-1 (Initia) commented:

This PR should fix that too, it will check balance and min gas prices at tx queue.

berndartmueller (warden) commented:

Great finding! I had this as a note in my initial review.

The impact is that users cannot queue transactions with a larger nonce than their current nonce. Sending a tx with the current nonce still works. Hence, only the ability to queue transactions is affected. No assets are lost/impacted, rather, the protocol's functionality is affected. Might be worth reconsidering the severity.

LSDan (judge) commented:

I initially allowed the high risk because of the ease of overloading the queue. But good point that users would still be able to execute single transactions. With this in mind, medium is a better fit.

EVM calls via the custom Cosmos precompile fail to refund the remaining gas in case of failure

Submitted by 0xAlix2

Severity: Medium

Finding description and impact

In the previous contest, F-11 was submitted, which discusses an issue where EVM calls via the custom Cosmos precompile fees were wrongly charged from the parent call/context. A fix was introduced to fix this, this was introduced where the provided gas limit is precharged when adding the "sub-message" to the context's messages to dispatch later.

https://github.com/initia-labs/minievm/pull/165/commits/c7afb0edc4f1535d15423a3707d953ac48328971#diff-0fd32004245b54738ecb05582a44de3626b866e4ad2245eb91a162c18baf97b7R296-R298:

```
// pre-charge the gas for the execute cosmos
```

```
ctx.GasMeter().ConsumeGas(executeCosmosArguments.GasLimit, "pre-charge
execute cosmos gas")
```

After the message is dispatched, the remaining is refunded.

https://github.com/initia-labs/minievm/blob/abe81cb5a813090837b182431d2b1261fede334a/x/evm/keeper/context.go#L570-L584:

```
if !success {
        // return error if failed and not allowed to fail
        if !allowFailure {
                return
        }

        // emit failed reason event if failed and allowed to fail
        event =
event.AppendAttributes(sdk.NewAttribute(types.AttributeKeyReason,
err.Error()))
} else {
        // commit if success
        commit()

        // refund remaining gas
@>      parentCtx.GasMeter().RefundGas(ctx.GasMeter().GasRemaining(),
"refund gas from submsg")
}
```

However, as can be seen, it only refunds the unused gas in case of success, which is wrong.

When a Solidity call is made with a gas limit, the call is expected to at most use X gas, however, because of the precharge here, it is expected to return the unused even if the call reverted/failed.

This breaks EVM compliance and causes a loss of funds to the caller.

Proof of Concept

Add the following in `minievm/x/evm/contracts/counter/Counter.sol` and run make contracts-gen:

```
function _revert() external pure {
    revert("revert reason dummy value for test");
}
```

```
function recursive_5() public {
    emit recursive_called(0);

    COSMOS_CONTRACT.execute_cosmos_with_options(
        string(
            abi.encodePacked(
                '{"@type": "/minievm.evm.v1.MsgCall",',
                '"sender": "',
                COSMOS_CONTRACT.to_cosmos_address(address(this)),
                '",',
                '"contract_addr": "',
                Strings.toHexString(address(this)),
                '",',
                '"input": "',
                Strings.toHexString(
                    abi.encodePacked(this._revert.selector)
                ),
                '",',
                '"value": "0",',
                '"access_list": []}'
            )
        ),
        uint64(10_000),
        ICosmos.Options(true, 0)
    );
}
```

Add the following in `minievm/x/evm/keeper/context.go`, just under:

```
        // and execute it
        res, err := handler(ctx, msg)
+       fmt.Println("Gas remaining after sub-call",
parentCtx.GasMeter().GasRemaining(), ctx.GasMeter().GasRemaining())
        if err != nil {
                return
        }
```

Add the following test in `minievm/x/evm/keeper/context_test.go`:

```
func Test_Call_NotRefundingOnFailure(t *testing.T) {
        ctx, input := createDefaultTestInput(t)
        _, _, addr := keyPubAddr()

        counterBz, err := hexutil.Decode(counter.CounterBin)
```

```
        require.NoError(t, err)

        // deploy counter contract
        caller := common.BytesToAddress(addr.Bytes())
        retBz, contractAddr, _, err := input.EVMKeeper.EVMCreate(ctx,
caller, counterBz, nil, nil)
        require.NoError(t, err)
        require.NotEmpty(t, retBz)
        require.Len(t, contractAddr, 20)

        // call recursive function
        parsed, err := counter.CounterMetaData.GetAbi()
        require.NoError(t, err)

        inputBz, err := parsed.Pack("recursive_5")
        require.NoError(t, err)

        // call with 1M gas limit
        ctx = ctx.WithGasMeter(storetypes.NewGasMeter(1_000_000))

        _, logs, err := input.EVMKeeper.EVMCall(ctx, caller,
contractAddr, inputBz, nil, nil)
        require.NoError(t, err)
        require.Equal(t, 1, len(logs))

        fmt.Println("Gas remaining after call",
ctx.GasMeter().GasRemaining())
}
```

Run and notice the logs:

```
=== RUN   Test_Call_NotRefundingOnFailure
Gas remaining after sub-call 883758 5658
Gas remaining after call 883758
```

The parent's gas remaining after the sub call is equal to that at the very end of the test, knowing that the child's remaining gas is `> 0`, but it wasn't refunded.

Recommended mitigation steps

Ensure that the remaining gas is refunded in case of failure:

```
        func (k Keeper) dispatchMessage(parentCtx sdk.Context, request
types.ExecuteRequest) (logs types.Logs, err error) {
```

```
                // ... snip ...
                defer func() {
                        // ... snip ...

                        success := err == nil

                        // create submsg event
                        event := sdk.NewEvent(
                                types.EventTypeSubmsg,

sdk.NewAttribute(types.AttributeKeySuccess, fmt.Sprintf("%v", success)),
                        )

                        if !success {
+                               // refund remaining gas
+
parentCtx.GasMeter().RefundGas(ctx.GasMeter().GasRemaining(), "refund gas
from submsg")
                                // return error if failed and not allowed
to fail
                                if !allowFailure {
                                        return
                                }

                                // emit failed reason event if failed and
allowed to fail
                                event =
event.AppendAttributes(sdk.NewAttribute(types.AttributeKeyReason,
err.Error()))
                        } else {
                                // commit if success
                                commit()

                                // refund remaining gas

parentCtx.GasMeter().RefundGas(ctx.GasMeter().GasRemaining(), "refund gas
from submsg")
                                // refund remaining gas

parentCtx.GasMeter().RefundGas(ctx.GasMeter().GasRemaining(), "refund gas
from submsg")
                        }

                        // reset error because it's allowed to fail
                        err = nil
```

```
                        // ... snip ...
                }()

                // ... snip ...
        }
```

Links to affected code

context.go#L570-L584

beer-1 (Initia) commented:

Fixed here.

Callback message's error is not propagated to the paren't process

Submitted by 0xAlix2

Severity: Medium

Finding description and impact

Users can dispatch EVM messages from the Cosmos precompile by calling one of the following functions in a Solidity contract:

```
COSMOS_CONTRACT.execute_cosmos_with_options(...)
COSMOS_CONTRACT.execute_cosmos(...)
```

When dispatching an EVM message, the user can optionally pass a callback ID, allowing a callback to be executed after the original message completes. In Solidity/EVM, if contract X calls Y, and Y calls back X in a "callback context," a callback revert causes the entire transaction to revert.

Similarly, if function F1 on contract X calls F2 on X, which calls contract Y, and Y callbacks `X–if F2 reverts`, F1 reverts; and if the callback reverts, everything reverts. However, this is not the case here: if the sub-message reverts, the original transaction correctly reverts, but if the sub-message's callback reverts, the original transaction is still committed, which is incorrect.

The error is just thrown away:

```
// if callback exists, execute it with parent context because it's
already committed
if callbackId > 0 {
        inputBz, err := k.cosmosCallbackABI.Pack("callback", callbackId,
success)
```

```
        if err != nil {
                return
        }

        var callbackLogs types.Logs
        _, callbackLogs, err = k.EVMCall(parentCtx, caller.Address(),
caller.Address(), inputBz, nil, nil)
@>      if err != nil {
                return
        }

        logs = append(logs, callbackLogs...)
}
```

This can lead to serious issues, especially in DeFi contracts, potentially causing loss of funds.

Proof of Concept

Apply the following diff in `minievm/x/evm/contracts/counter/Counter.sol`, and run `make contracts-gen`:

```
-    function callback(uint64 callback_id, bool success) external {
-        emit callback_received(callback_id, success);
-    }
+
+    event callback_received_2(
+        uint64 callback_id,
+        bool success,
+        uint64 gas_received
+    );
+
+    function callback(uint64 callback_id, bool success) public {
+        emit callback_received_2(callback_id, success,
uint64(gasleft()));
+        revert("revert reason dummy value for test");
+    }
+
+    function noop_2() public pure {}
+
+    function recursive_7() public {
+        emit recursive_called(7);
+
+        COSMOS_CONTRACT.execute_cosmos_with_options(
+            string(
+                abi.encodePacked(
```

```
+                    '{"@type": "/minievm.evm.v1.MsgCall",',
+                    '"sender": "',
+                    COSMOS_CONTRACT.to_cosmos_address(address(this)),
+                    '",',
+                    '"contract_addr": "',
+                    Strings.toHexString(address(this)),
+                    '",',
+                    '"input": "',
+
Strings.toHexString(abi.encodePacked(this.noop_2.selector)),
+                    '",',
+                    '"value": "0",',
+                    '"access_list": []}'
+                )
+            ),
+            uint64(200_000),
+            ICosmos.Options(true, 1)
+        );
+    }
```

Add the following test in `minievm/x/evm/keeper/context_test.go`:

```
func Test_Call_CallbackErrorNotPropagated(t *testing.T) {
        ctx, input := createDefaultTestInput(t)
        _, _, addr := keyPubAddr()

        counterBz, err := hexutil.Decode(counter.CounterBin)
        require.NoError(t, err)

        // deploy counter contract
        caller := common.BytesToAddress(addr.Bytes())
        retBz, contractAddr, _, err := input.EVMKeeper.EVMCreate(ctx,
caller, counterBz, nil, nil)
        require.NoError(t, err)
        require.NotEmpty(t, retBz)
        require.Len(t, contractAddr, 20)

        // call recursive function
        parsed, err := counter.CounterMetaData.GetAbi()
        require.NoError(t, err)

        inputBz, err := parsed.Pack("recursive_7")
        require.NoError(t, err)
```

```
        _, logs, err := input.EVMKeeper.EVMCall(ctx, caller,
contractAddr, inputBz, nil, nil)
        require.NoError(t, err) // no error returned
        require.Equal(t, 1, len(logs))
}
```

Recommended mitigation steps

Ensure that the callback message's error is propagated to the parent's context, and if an error exists revert all changes done, following what a Solidity contract call would expect. Or, allow users to "allow callback failures", similar to what's provided in `execute_cosmos`'s `allow_failure` option, by having something like `allow_callback_failure`, this gives the users more control over wether they are okay with it reverting or not.

Links to affected code

context.go#L594-L597

beer-1 (Initia) commented:

Fixed here.

a_kalout (warden of team 0xAlix2) commented:

I respectfully believe this is of high severity because the lack of proper error propagation in callbacks breaks core EVM expectations and can lead to critical vulnerabilities in DeFi protocols.

For example, imagine a lending protocol that uses a Cosmos message to seize collateral during liquidation, and relies on a callback to transfer the seized funds to the liquidator. If the callback fails but the error is ignored, the protocol thinks the transfer succeeded, but the funds are stuck. The liquidator loses money, and the system ends up in an inconsistent state.

In normal EVM behavior, this would revert the whole transaction. Silently swallowing the error breaks that expectation and can lead to real financial loss.

LSDan (judge) commented:

Ruling stands. In all of the scenarios described, including the comment above, medium is a better fit.

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit

submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.