Audit Report

Filecoin EVM (FEVM)

v1.1

March 9, 2023

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

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

Introduction

Purpose of This Report

Oak Security has been engaged by Filecoin Foundation to perform a security audit of the Filecoin EVM (FEVM).

The objectives of the audit are as follows:

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

This report represents a summary of the findings.

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

Codebase Submitted for the Audit

The audit has been performed over the course of 3 two-week sprints on the following GitHub repositories:

`builtin-actors`: https://github.com/filecoin-project/builtin-actors (only the `actors/evm` and `actors/eam` directories were in the scope of this audit)

`ref-fvm`: https://github.com/filecoin-project/ref-fvm

Commit hash for the `builtin-actors` repository:
`1b11df4b399550753a4105f45f58bc07015af2a3`

Commit hash for the `ref-fvm` repository:
`31babafe2b96dce6a8ba33ac23e54ce30ee16881`

Sprint #1

1. EVM runtime actor (EVM implementation, precompiles, correctness of EVM opcodes)
   https://github.com/filecoin-project/builtin-actors/actors/evm

Sprint #2

1. EVM runtime actor
2. EAM (Ethereum Address Manager)
   https://github.com/filecoin-project/builtin-actors/actors/eam
3. FVM enhancements
   a. Gas model
   b. F4 addresses (in both FVM and FEVM)

Sprint #3

1. Reviewing the message execution flow (in regards to error handling and memory limits) and reviewing the kernel setup
2. Wasmtime integration
3. FVM logs

Methodology

The audit has been performed in the following steps:

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

Functionality Overview

The Filecoin EVM (FEVM) is an Ethereum Virtual Machine (EVM) compatible, virtualized runtime on top of the Filecoin Virtual Machine (FVM).

How to Read This Report

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

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

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

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

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

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

Code Quality Criteria

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

Criteria	Status	Comment
Code complexity	**Medium**	Many deep interactions between the FEVM and the FVM.
Code readability and clarity	**Medium-High**	-
Level of documentation	**High**	-
Test coverage	**Low-Medium**	18.45% test coverage for EVM actor. 3.95% test coverage for EAM actor. 47.67% test coverage for ref-fvm.

Summary of Findings

No	Description	Severity	Status
1	Randomness precompile does not charge gas for external call processing	Critical	Resolved
2	Actor installation is not charging gas on wasm preload error	Critical	Acknowledged
3	No gas is charged for event emission and storage	Critical	Acknowledged
4	`DIFFICULTY` opcode is not following the EVM specification	Major	Acknowledged
5	Storage opcodes do not follow EIP-2929	Major	Acknowledged
6	Maximum stack size can be exceeded	Major	Resolved
7	Deviating from the EVM, retrieving the code for non-existing accounts reverts the transaction	Major	Resolved
8	Opcode gas accounting deviating from the Ethereum EVM specification could lead to unexpected behavior	Major	Acknowledged
9	`assert` instruction does not consume all the remaining gas as defined in the EVM specification	Major	Acknowledged
10	The bitwise `SAR` opcode returns 1 instead of 0 if `shift` is greater than or equal to 256	Major	Resolved
11	`LOG` opcodes do not revert in case of a `STATICCALL`	Major	Resolved
12	`block.coinbase` is not following the EVM specification	Major	Acknowledged
13	FEVM gas consumption could saturate the available gas in a block and slow down the processing of core Filecoin operations	Major	Acknowledged
14	The `CREATE2` opcode smart contract generation is not supported by FEVM	Major	Resolved
15	FEVM allows funds to be sent to a contract without the need for a `receive` or `fallback` method	Major	Acknowledged
16	After `SELFDESTRUCT`, FEVM does not follow the EVM specification	Major	Resolved

17	DELEGATECALL does not propagate msg.value to the implementation contract	Major	Resolved
18	Diverging behavior of gas refunds during SELFDESTRUCT between EVM and FEVM	Major	Acknowledged
19	Retrieving the code of precompile addresses reverts, which diverges from the EVM specification	Minor	Resolved
20	Incrementing the nonce of an account with a nonce of u64::MAX panics with an overflow error	Minor	Acknowledged
21	The GASPRICE opcode does not return the effective gas price	Minor	Resolved
22	FEVM CALL instruction computational overhead differs slightly from standard EVM	Minor	Acknowledged
23	EVM actor system.nonce is not reverted if EAM CREATE or CREATE2 messages fail	Minor	Acknowledged
24	Processing transfer of funds during SELFDESTRUCT does not charge gas	Minor	Acknowledged
25	A malformed and unserializable event leads to a panic in the actor	Minor	Acknowledged
26	Miner is not penalized for messages with an improper gas limit	Minor	Acknowledged
27	Ethereum addresses in the range between 0x0 and 0xffff are considered precompiles	Informational	Resolved
28	FEVM precompiles can potentially conflict with future EVM precompiles	Informational	Resolved
29	Misleading comment for wasm memory copy costs	Informational	Acknowledged
30	MachineExecRet struct is unnecessarily defined in execute_message scope during its execution	Informational	Acknowledged
31	Missing validation for the desired param during wasm memory_growing	Informational	Acknowledged
32	Redundant balance check when transferring funds	Informational	Acknowledged
33	An implicit message with an improper gas limit set could fail with insufficient gas	Informational	Acknowledged
34	Message calls without input data are implicitly converted to plain value transfers and do not execute code	Informational	Resolved

Detailed Findings

1. Randomness precompile does not charge gas for external call processing

Severity: Critical

The `get_randomness` precompile defined in `builtin_actors/actors/evm/src/interpreter/precompiles.rs:346`, which is callable from the FEVM, does not account for gas for the entire process.

In `ref_fvm/fvm/src/gas/price_list.rs:684`, gas is only charged for hashing and the precompile call but not for the random value generation process.

Since this function can be invoked by any contract in FEVM, an attacker is able to create a contract that intensively calls this precompile in order to use computational resources without being charged appropriate gas. This could lead to overloading nodes in the network, up to the point where block production slows down or even stops due to timeouts being reached.

Recommendation

We recommend charging gas for both `ticket` and `beacon` random value generation.

Status: Resolved

The client has disabled the `get_randomness` precompile.

2. Actor installation is not charging gas on wasm preload error

Severity: Critical

In `ref-fvm/fvm/src/kernel/default.rs:906`, the `install_actor` function is charging gas after the `engine` preload operations.

This implies that if the `engine` fails to set up the wasm engine and bytecode, it will return an error interrupting the execution without calling the `on_install_actor` function, which is responsible for charging gas.

An attacker could take advantage of this by loading an invalid serialized wasm file that consumes computational resources and then returns an error in the preload operations paying no additional gas.

Recommendation

We recommend charging gas also if a preloading error occurs.

Status: Acknowledged

The client acknowledged the finding and pointed out that users are not allowed to install new actors in the current version. It will be fixed in the new `m2` version.

3. No gas is charged for event emission and storage

Severity: Critical

In `fvm/src/gas/price_list.rs:274-281`, the gas cost for event-related operations is zero.

This implies that no additional gas cost is charged to actors that emit events, including FEVM.

Additionally, during the `execute_message` function, the `commit_events` function, defined in `fvm/src/machine/default.rs`, performs an unbounded loop through all the provided `StampedEvents` in order to store them one by one in an Array Mapped Trie without accounting for gas.

Since no gas is charged for event emission and storage, an attacker could leverage this behavior in order to consume computational resources without paying additional gas. This can be exploited to slow down block production or even halt the chain.

Recommendation

We recommend charging gas for event emissions and committing them as a bulk instead of iterating through them.

Status: Acknowledged

4. `difficulty` opcode is not following the EVM specification

Severity: Major

As per the legacy EVM specification, the `DIFFICULTY` opcode returns the current block difficulty. After the introduction of EIP-4399, this opcode behavior changed slightly in order to return an `RNG` beacon following the `RANDAO` specification.

The current FEVM implementation of the `DIFFICULTY` opcode always returns zero. This implies that smart contracts using this functionality could suffer unexpected behaviors.

Recommendation

We recommend implementing the EVM specification for the `DIFFICULTY` opcode.

Status: Acknowledged

The client replaced the `DIFFICULTY` opcode with the `PREVRANDAO` opcode.

5. Storage opcodes do not follow EIP-2929

Severity: Major

In the EVM Berlin fork, EIP-2929 got introduced, which is not implemented in current storage opcodes in the FEVM. Because of that, the FEVM implementation diverts from the gas handling of the EVM.

Recommendation

We recommend either adhering to EIP-2929 or explicitly communicating in the developer documentation of the FEVM that the FEVM does not honor EIP-2929 (stating that FEVM accounts for the gas consumption in wasm).

Status: Acknowledged

6. Maximum stack size can be exceeded

Severity: Major

Prior to executing an opcode, the stack size is checked by calling the `Stack::ensure` function in `builtin-actors/actors/evm/src/interpreter/stack.rs`. If the current stack size is insufficient, the stack size is doubled, possibly increasing the stack size beyond `STACK_SIZE`.

If the `Stack::ensure` function is called again as part of executing the next opcode, and the required size is less or equal to the current stack size, it will consider the stack size sufficient without checking the size limit. This leads to the stack size being increased beyond the intended limit of `1024` to a maximum stack size of `2048`.

Recommendation

We recommend checking the stack size limit before checking if the stack size is sufficient.

Status: Resolved

7. Deviating from the EVM, retrieving the code for non-existing accounts reverts the transaction

Severity: Major

The opcodes `EXTCODESIZE`, `EXTCODECOPY`, and `EXTCODEHASH` retrieve the bytecode by calling the `get_evm_bytecode_cid` function in `builtin-actors/actors/evm/src/interpreter/instructions/ext.rs`. However, this function returns an unhandled error in case the address belongs to a non-existent account, leading to the transaction being reverted.

On the contrary, the EVM returns `0` for `EXTCODESIZE`, and the keccak256 hash of empty data (`0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470`) for `EXTCODEHASH` if the target address is a non-existent account.

Recommendation

We recommend adhering to the EVM specification and handling returned errors from the `get_evm_bytecode_cid` function.

Status: Resolved

8. Opcode gas accounting deviating from the Ethereum EVM specification could lead to unexpected behavior

Severity: Major

Due to the different gas accounting, contracts developed for the EVM might experience compatibility and portability issues when deployed to the FEVM.

Commonly, code is optimized regarding gas cost, but since the FEVM's gas accounting is different, the same contract executing on the EVM might run out of gas on FEVM or vice versa.

Please find below a few concrete examples of issues caused by the diverging gas accounting:

1. In the following example, the `store` function is executable on Ethereum and Binance Smart Chain with `num = 50_000`, while not executable on the FEVM. It runs out of gas if `num >= 45_000`.

```
uint256 number;

function store(uint256 num) public {
    for(uint256 i = 0; i<num; i++){
        number = i;
    }
```

```
        }
```

2. If a contract contains a `call` instruction with an explicitly provided gas limit via `{gas:value}`, it may not be portable from the EVM to the FEVM.

   The optionally defined gas limit for the `call` instruction has to account for all the internal FVM operations (e.g., `system.send` handling), making it complex for developers to guess the correct amount of gas limit to provide.

   An example is provided in Appendix 1.

3. Ethereum smart contract developers usually optimize the code and the build process to minimize gas costs for the user.

   `solc` enables different types of optimizations that rely on opcodes https://docs.soliditylang.org/en/v0.8.17/internals/optimizer.html.

   Using the `--optimize-runs` compiler flag optimizes the gas cost of contracts designed to be deployed once and executed multiple times or if they are children of a factory contract.

   Also, some common patterns exist to optimize the code to use cheaper opcodes. For example, using `SSTORE` to store zeros in the EVM is cheaper than other numbers.

   All of these types of optimizations are not effective in the FEVM and could lead to portability issues.

   An example of how optimizations affect gas usage in FEVM is provided in Appendix 2.

Recommendation

We recommend using a gas mapping mechanism that would enable the FEVM to calculate the gas cost internally, in the standard EVM way, and then convert it in the FVM gas convention.

Status: Acknowledged

9. `assert` instruction does not consume all the remaining gas as defined in the EVM specification

Severity: Major

As per the EVM specification, the `assert` instruction should consume all the remaining available gas by executing the `0xFE INVALID` opcode.

However, the FEVM implementation returns error code 33 without charging the remaining gas.

Recommendation

We recommend implementing the EVM specification for the `assert` instruction in order to consume all the sender-provided gas.

Status: Acknowledged

10. The bitwise `SAR` opcode returns 1 instead of 0 if `shift` is greater than or equal to 256

Severity: Major

The EVM specification states that the `SAR` opcode should return `0` if `shift` is greater than or equal to `256` for a `value >= 0`. However, the FEVM implementation in `builtin-actors/actors/evm/src/interpreter/instructions/bitwise.rs:43` for the `SAR` opcode returns `U256::ONE` (i.e. `1`) instead of `0` in this case. This can lead to incorrect results for the `SAR` opcode.

Recommendation

We recommend returning `0` instead of `1` in the `SAR` opcode implementation if `shift` is greater than or equal to `256` to comply with the EVM specification.

Status: Resolved

11. `LOG` opcodes do not revert in case of a `STATICCALL`

Severity: Major

According to the EVM specification, executing `LOG` opcodes is not allowed in a read-only `STATICCALL` and should revert in this case. However, the FEVM `log` function implementing the `LOG` opcodes in `builtin-actors/actors/evm/src/interpreter/instructions/log.rs` does not check if the current call is a `staticcall`.

Recommendation

We recommend returning an error if the `LOG` opcode is used within a read-only `STATICCALL`.

Status: Resolved

12. `block.coinbase` is not following the EVM specification

Severity: Major

According to the EVM specification, the `block.coinbase` instruction should return the miner of the current block. In contrast, in the FEVM, a zero value is returned.

This implies that all the smart contracts that use this information, for example, as a source of entropy, will not work as intended.

Examples of usages of this instruction on open source projects can be found with this GitHub query, resulting in around 24K results.

Recommendation

We recommend implementing the EVM specification for the `block.coinbase` value.

Status: Acknowledged

13. FEVM gas consumption could saturate the available gas in a block and slow down the processing of core Filecoin operations

Severity: Major

The FEVM gas consumption is generally higher than that of other core Filecoin network transactions. For example, a `FIL` transfer performed through FEVM costs `1.651.058` gas, while the same operation costs `489.268` with a base `Send` transaction.

Since FEVM-related gas fees are on a different numeric magnitude than base operations, this could lead to a situation where FEVM transactions could saturate the block, leaving little room for core Filecoin operations.

Recommendation

We recommend defining priority and segregation mechanisms in order to make protocol core operations privileged with respect to FVM and FEVM ones.

Status: Acknowledged

The client acknowledged the finding but decided to not address it for now as this is how it was initially designed.

14. The `CREATE2` opcode smart contract generation is not supported by FEVM

Severity: Major

The FEVM does not support smart contract creation using the `CREATE2` opcode, while the EVM does. This behavior breaks the compatibility of smart contracts expected to work in the FEVM.

A sample smart contract used for testing is available in Appendix 3. The functions `create2NewToken` and `deploy` in the provided smart contract revert when executed with the following error:

```
{
  "code": 1,
   "message": "message execution failed: exit 33, reason:
message failed   with backtrace:\n00: f01306 (method 2) --
contract reverted (33)\n01: f010 (method 3) -- Serialization
error for Cbor protocol: Mismatch { expect_major: 2, byte:
152 } (21)\n (RetCode=33)"
}
```

As a reference, the same smart contract can be successfully deployed with the `CREATE2` opcode on the Goerli network:

https://goerli.etherscan.io/tx/0x60150e92677fc2738571222de5a0c107780c131a38f8ff7a493f258a56444f93

Recommendation

We recommend adding support for the `CREATE2` opcode.

Status: Resolved

15. FEVM allows funds to be sent to a contract without the need for a `receive` or `fallback` method

Severity: Major

Using the `send` and `transfer` functions on the FEVM, a contract can receive funds even without having a `fallback` or `receive` function. In contrast, the EVM specification requires that a smart contract cannot receive payments if it lacks a `receive` or `fallback` function. Breaking this assumption may result in unexpected behavior, including potentially lost funds.

A sample smart contract used for testing is available in Appendix 4.

Recommendation

We recommend following the EVM specification and disallowing payments to contracts that do not have a `receive` or `fallback` method.

Status: Acknowledged

16. After `SELFDESTRUCT`, FEVM does not follow the EVM specification

Severity: Major

In the EVM, when a smart contract executes the `SELFDESTRUCT` opcode, it thereafter operates as an EOA, which means that its code is erased while it continues to receive payments. However, the funds are no longer available since smart contracts do not have explicit private keys. In the present implementation of the FEVM, after executing `SELFDESTRUCT` on a smart contract, payments can no longer be received, and the error *"actor doesn't exist"* is returned.

This behavior of FEVM deviates from the EVM, which may cause problems when migrating EVM-based smart contracts directly to FEVM.

A sample smart contract used for testing is available in Appendix 5.

Recommendation

We recommend following the EVM specification to achieve greater portability.

Status: Resolved

17. `DELEGATECALL` does not propagate `msg.value` to the implementation contract

Severity: Major

As per the specification of the `DELEGATECALL` opcode, the execution context should be the same as the caller contract, which implies that the `msg.value` and `msg.sender` values should be the same in the implementation contract, and the implementation contract can utilize those values during execution. However, the FEVM implementation of `DELEGATECALL` does not propagate the `msg.value` of the caller context during the delegatecall, since `msg.value` is passed as `U256::zero()` in `builtin-actors/actors/evm/src/interpreter/instructions/call.rs:148`. This reduces the portability of EVM contracts to the FEVM and may even lead to unexpected behavior with potentially catastrophic consequences if teams are unaware of these divergences in behavior.

A sample smart contract used for testing is available in Appendix 6.

Recommendation

We recommend passing the given value of `msg.value` in the `DELEGATECALL` opcode, similar to the `CALL` opcode in `builtin-actors/actors/evm/src/interpreter/instructions/call.rs:129`.

Status: Resolved

18. Diverging behavior of gas refunds during `SELFDESTRUCT` between EVM and FEVM

Severity: Major

According to the EVM specification, the `SELFDESTRUCT` opcode refunds `24000` gas units. The accumulated refund can not exceed half of the gas used for the current context, while a gas refund is provided only once at the end of the transaction's execution. In contrast, the FEVM refunds gas upon `SELFDESTRUCT` during the execution of the transaction in `ref-fvm/fvm/src/kernel/default.rs:180`. That implies that the execution can use refunded gas, which may lead to unexpected behavior when porting smart contracts from the EVM to the FEVM.

Recommendation

We recommend adhering to the same behavior as the EVM `SELFDESTRUCT` or highlighting the divergence in the documentation and other educational materials for developers.

Status: Acknowledged

19. Retrieving the code of precompile addresses reverts, which diverges from the EVM specification

Severity: Minor

In the FEVM, the opcodes `EXTCODESIZE`, `EXTCODECOPY`, and `EXTCODEHASH` revert if the target address is a precompile. Given that the FEVM considers all addresses in the range between `0x0` and `0xffff` as precompiles, the aforementioned opcodes revert for those addresses even if they are not precompiles. See `builtin-actors/actors/evm/src/interpreter/instructions/ext.rs:59`.

On the contrary, the EVM returns `0` for `EXTCODESIZE` and the keccak256 hash of empty data (`0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470`) for `EXTCODEHASH` if the target address is a precompile.

Recommendation

We recommend adhering to the EVM specification.

Status: Resolved

20. Incrementing the nonce of an account with a nonce of `u64::MAX` panics with an overflow error

Severity: Minor

The opcodes `CREATE` and `CREATE2` increment the nonce of the sender account by `1` in `builtin-actors/actors/evm/src/interpreter/instructions/lifecycle.rs:66` and `102`. However, if the nonce of the sender account is `u64::MAX`, the nonce will overflow and panic with an error. This is in contrast to the EVM implementation in the Geth (Go Ethereum) client, which gracefully handles an overflow of the nonce and pushes `0` on the stack.

It is very unlikely that an account will have a nonce of `u64::MAX`. However, it might be possible that the nonce is set to `u64::MAX` for self-destroyed contracts in a future version of the FEVM.

Recommendation

We recommend checking if the nonce is `u64::MAX` and returning early with a value of 0, in line with the Geth implementation.

Status: Acknowledged

21. The `GASPRICE` opcode does not return the effective gas price

Severity: Minor

EIP-1559 states that the `GASPRICE` opcode must return the `effective_gas_price` as defined in the specification. The effective gas price is the sum of the base fee per gas and the priority fee per gas, capped to the maximum `max_fee_per_gas`.

However, the FEVM implementation for the `GASPRICE` opcode in `builtin-actors/actors/evm/src/interpreter/instructions/context.rs:84` does not return the capped value. Instead, it returns the sum of the base fee and the priority fee per gas.

Recommendation

We recommend returning the capped value to comply with EIP-1559.

Status: Resolved

22. FEVM `CALL` instruction computational overhead differs slightly from the EVM

Severity: Minor

In the FEVM, the execution of `CALL` instructions consumes more resources compared to the EVM. This is caused by the various FVM operations required to set up the environment.

An example with benchmarks is available in Appendix 7.

This could lead to contract portability issues since, in some cases, this would result in a different gas consumption with respect to the EVM.

Recommendation

We recommend charging a defined and static amount of gas similar to the EVM for the `CALL` environment setup.

Status: Acknowledged

23. EVM actor `system.nonce` is not reverted if EAM `CREATE` or `CREATE2` messages fail

Severity: Minor

In `actors/evm/src/interpreter/instructions/lifecycle.rs:115`, the `create_init` function sends a message from the EVM actor to the EAM actor to trigger the `CREATE` method and updates the EVM actor's incremental `system.nonce` in `actors/evm/src/interpreter/instructions/lifecycle.rs:66`.

However, in case the execution fails to send the message, the `system.nonce` is not reverted to the previous value.

This would lead to an incorrect `nonce` ordering in the `EVM` actor since it is incrementing it also if the transaction is not sent.

Recommendation

We recommend handling the error and reverting the nonce increase.

Status: Acknowledged

The client decided to increment the nonce even if the contract creation fails.

24. Processing transfer of funds during `SELFDESTRUCT` does not charge gas

Severity: Minor

In `ref-fvm/fvm/src/kernel/default.rs:197`, there is no gas charged while transferring funds from `actor_id` to `beneficiary_id` during `SELFDESTRUCT`. This allows an attacker to overload a node with computation at minimal cost.

Recommendation

We recommend charging gas while transferring funds during `SELFDESTRUCT`.

Status: Acknowledged

25. A malformed and unserializable event leads to a panic in the actor

Severity: Minor

The FEVM emits events via the FVM SDK function `emit_event` in `ref-fvm/sdk/src/event.rs:8`. Events are then serialized to CBOR format. However, if the event is malformed, the serialization fails and panics the actor due to the usage of `expect`.

Recommendation

We recommend propagating the error rather than panicking.

Status: Acknowledged

26. Miner is not penalized for messages with an improper gas limit

Severity: Minor

The `DefaultExecutor::preflight_message` function in `ref-fvm/fvm/src/executor/default.rs` does a preliminary check of the message before further processing. Message validity is checked with the `Message::check` function in `ref-fvm/shared/src/message.rs:32`. The `check` function validates the message gas limit. However, if the check fails, the message fails without penalizing the miner.

The Lotus client presently avoids propagation of such failing messages. We still classify this issue as minor, since there could be other client implementations not verifying the gas limit, which could lead to possible attacks on miners in the future.

Recommendation

We recommend penalizing the miner for such invalid messages.

Status: Acknowledged

The client acknowledged the finding and pointed out that the FVM expects the node clients (e.g., Lotus) to perform this check and penalize dishonest miners appropriately.

27. Ethereum addresses in the range between 0x0 and 0xffff are considered precompiles

Severity: Informational

The FEVM implementation in `builtin-actors/actors/evm/src/interpreter/address.rs:41` considers an address a precompile if the address is in the range between `0x00` and `0xffff`, without verifying whether the address is an actual precompile. Addresses incorrectly classified as precompiles can therefore not be used as regular EOA or smart contract addresses holding and receiving funds.

Recommendation

We recommend verifying that the address is a precompile before considering it as one.

Status: Resolved

28. FEVM precompiles can potentially conflict with future EVM precompiles

Severity: Informational

Currently, the addresses from `0x01` to `0x09` are used by the EVM precompiles. The FEVM adds additional custom precompiles in the range between `0x0a` and `0x0e`. However, if the EVM specification adds additional precompiles, the FEVM precompiles might conflict with such new EVM precompiles.

Recommendation

We recommend reserving the address space from `0x00` to `0xff` for EVM precompiles and adding custom FEVM precompiles starting from `0x100`.

Status: Resolved

29. Misleading comment for wasm memory copy costs

Severity: Informational

The comment in `ref-fvm/fvm/src/gas/price_list.rs:265` states:

```
// Don't yet charge anything for copying.
```

However, there is a cost per byte of `Gas::from_milligas(400)` for copying memory.

Recommendation

We recommend updating the comment to reflect the actual cost.

Status: Acknowledged

30. `MachineExecRet` struct is unnecessarily defined in `execute_message` scope during its execution

Severity: Informational

In `fvm/src/executor/default.rs:74`, during the execution of the `execute_message` function, the `MachineExecRet` struct is defined each time that the function is executed.

Since this struct does not include any closures over variables defined in its context, it could be defined globally outside the function scope.

Recommendation

We recommend defining the `MachineExecRet` struct outside of the function body.

Status: Acknowledged

31. Missing validation for the `desired` param during wasm `memory_growing`

Severity: Informational

As per the [wasmtime documentation](), the `desired` and `current` parameter values should be a multiple of the WASM page size. However, appropriate validation is missing in `filecoin-project/ref-fvm/fvm/src/machine/limiter.rs:52`.

Recommendation

We recommend adding validation for the `desired` parameter value to ensure a multiple of WASM page size is used.

Status: Acknowledged

32. Redundant balance check when transferring funds

Severity: Informational

The `DefaultMachine::transfer` function in `ref-fvm/fvm/src/machine/default.rs` checks the sender's balance before transferring funds. This check is not necessary, as the `from_actor.deduct_funds` function already checks the balance. The check is redundant and can be removed.

Recommendation

We recommend removing the duplicate balance check in the `transfer` function.

Status: Acknowledged

33. An implicit message with an improper gas limit set could fail with insufficient gas

Severity: Informational

Implicit messages (i.e. `ApplyKind::Implicit`) are internal messages for calls to the FVM. No gas is expected to be charged for these messages, but it is still accounted for in the current implementation. However, if the `msg.gas_limit` is not sufficiently set (i.e. to the maximum value of `i64`), the message could fail with insufficient gas as gas is charged for the message execution.

We classify this issue as informational since only the FVM client, i.e. Lotus, can send `Implicit` messages and set their `gas_limit`.

Recommendation

We recommend setting the `msg.gas_limit` to a default maximum gas limit in the `DefaultExecutor::execute_message` function or preventing the implicit message from being charged gas.

Status: Acknowledged

The client acknowledged the finding and pointed out that the FVM expects the node clients (e.g., Lotus) to perform this check.

34. Message calls without input data are implicitly converted to plain value transfers and do not execute code

Severity: Informational

The `Filecoin EVM compatibility specification` states the following:

Ethereum transactions carrying no input data are presumed to be Ethereum value transfers. These are converted to Filecoin sends (by setting method number = 0), and unlike in Ethereum, they have no opportunity to trigger smart contract logic. Note that such messages may carry value 0.

This deviates from the EVM specification and may lead to unexpected behavior. Please find two examples below:

1. The Solidity `.send(..)` and `.transfer(..)` functions forward a gas stipend of `2300` to the recipient address to prevent the receiver contract from using costly operations (for instance modifying the storage). If the receiver contract spends more than the provided gas stipend, the EVM reverts the transaction with an `OutOfGas` error.
   However, in the FEVM implementation, the transaction always succeeds and FIL is sent to the recipient address without executing any code from the recipient.

2. Upgradable proxy contracts may require a fallback function to be executed. However, the FEVM does not execute the code if the call to the recipient contract address does not contain any input data.

Recommendation

We recommend adhering to the EVM specification by executing any code from the recipient contract and reverting the transaction with an `OutOfGas` error if the recipient contract consumes more gas than the provided gas limit.

Status: Resolved

Appendix A: Test Cases

1. **Test case for "Opcodes gas accounting deviating from the Ethereum EVM specification could lead to unexpected behavior within contracts"**

In the two example contracts below, the `foo` function only executes without reverting if the provided `gas` is greater than 2,000,000.

```
contract Receiver {
    event Received(address caller, uint256 amount, uint256 gas, string
message);

    function foo(string memory _message) public payable returns
(uint256) {
        emit Received(msg.sender, msg.value, gasleft(), _message);
        return 1;
    }
}

contract Caller {
    event Response(bool success, bytes data);

    function testCallFoo(address payable _addr, uint256 gas) public
payable {
        (bool success, bytes memory data) = _addr.call{
            value: msg.value,
            gas: gas
        }(abi.encodeWithSignature("foo(string)", "call foo"));

        emit Response(success, data);
    }
}
```

2. Test case for “Opcodes gas accounting deviating from the Ethereum EVM specification could lead to unexpected behavior within contracts”

Below is the test code run for differential analysis of the gas optimization in FEVM compared to EVM.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

// GasMetering
contract GasMetering {
    // start - 50908 gas
    // use calldata - 49163 gas
    // load state variables to memory - 48952 gas
    // short circuit - 48634 gas
    // loop increments - 48244 gas
    // cache array length - 48209 gas
    // load array elements to memory - 48047 gas
    // uncheck i overflow/underflow - 47309 gas

    uint public total;

    // start - not gas optimized
    function sumIfEvenAndLessThan999(uint[] memory nums) external {
        for (uint i = 0; i < nums.length; i += 1) {
            bool isEven = nums[i] % 2 == 0;
            bool isLessThan99 = nums[i] < 99;
            if (isEven && isLessThan99) {
                total += nums[i];
            }
        }
    }

    // gas optimized
    // [1, 2, 3, 4, 5, 100]
    function sumIfEvenAndLessThan99(uint[] calldata nums) external {
        uint _total = total;
        uint len = nums.length;

        for (uint i = 0; i < len; ) {
            uint num = nums[i];
            if (num % 2 == 0 && num < 99) {
                _total += num;
            }
            unchecked {
                ++i;
            }
        }

        total = _total;
    }
}
```

As a consequence, after optimization, EVM consumes 40% less gas, whereas FEVM consumes 13% less gas.

Since many smart contracts may not be transferrable to the FEVM from EVM because their gas consumption is significantly higher than expected during the construction of the efficient smart contract.

3. Test case for “The create2 opcode smart contract generation is not supported by FEVM”

The code below has been used to check the functionality of `create2` opcode in FEVM. `create2NewToken` and `deploy` function both revert in FEVM while they work on the Goerli testnet.

```
pragma solidity 0.8.7;

// OZ code ref-
https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contr
acts

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

contract MyToken is ERC20, Ownable {
    event LogA(address);
    constructor(string memory _name, string memory _symbol) ERC20(_name,
_symbol) payable {
        emit LogA(address(this));
    }

    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }
}

contract MyTokenFactory {
    event NewTokenCreated(address token, address creator, string
typeOfCreation);

    constructor() {}

    function create2NewToken(bytes32 _salt) public payable {
        address tok =  address(new MyToken{salt: _salt, value:
msg.value}());
        emit NewTokenCreated(tok, msg.sender, "create2");
    }

}

// or
```

```
contract MyTokenFactory {
    event NewTokenCreated(address indexed token, address indexed
creator, string indexed typeOfCreation);

    event Deployed(address addr, uint salt);

    uint256 public testvalue;

    constructor() {}

    function getBytecode(string memory tokenName, string memory
tokenSymbol) public pure returns (bytes memory) {
        bytes memory bytecode = type(MyToken).creationCode;

        return abi.encodePacked(bytecode, abi.encode(tokenName,
tokenSymbol));
    }

    function getAddress(
        bytes memory bytecode,
        uint _salt
    ) public view returns (address) {
        bytes32 hash = keccak256(
            abi.encodePacked(bytes1(0xff), address(this), _salt,
keccak256(bytecode))
        );

        // NOTE: cast last 20 bytes of hash to address
        return address(uint160(uint(hash)));
    }

    function deploy(string memory tokenName, string memory tokenSymbol,
uint _salt) public payable {
        address addr;
        bytes memory bytecode = getBytecode(tokenName, tokenSymbol);
        /*
        NOTE: How to call create2

        create2(v, p, n, s)
        create new contract with code at memory p to p + n
        and send v wei
        and return the new address
        where new address = first 20 bytes of keccak256(0xff +
address(this) + s + keccak256(mem[p...(p+n)))
              s = big-endian 256-bit value
        */
        assembly {
            addr := create2(
                callvalue(), // wei sent with current call
                // Actual code starts after skipping the first 32 bytes
                add(bytecode, 0x20),
                mload(bytecode), // Load the size of code contained in
the first 32 bytes
                _salt // Salt from function arguments
            )

            if iszero(extcodesize(addr)) {
                revert(0, 0)
            }
        }

        emit Deployed(addr, _salt);
    }
}
```

4. Test case for “FEVM allows funds to be sent into the contract without the need for a receive or fallback method”

`NotReceiveEther` contract should not receive funds as per the EVM spec because it has neither a `fallback` nor a `receive` function. In the FEVM, it receives funds though. This makes the FEVM non-compatible with the EVM.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

contract NotReceiveEther {
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}

contract SendEther {
    function sendViaSend(address payable _to) public payable {
        // Send returns a boolean value indicating success or failure.
        // This function is not recommended for sending Ether.
        bool sent = _to.send(msg.value);
        require(sent, "Failed to send Ether");
    }

    function sendViaTransfer(address payable _to) public payable {
        // This function is no longer recommended for sending Ether.
        _to.transfer(msg.value);
    }
}
```

5. **Test case for “After selfdestruct, FEVM does not follow the same smart contract behavior as EVM-based smart contracts”**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

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

contract MyToken is ERC20, Ownable {
    event LogA(address);
    constructor(string memory _name, string memory _symbol) ERC20(_name,
_symbol) payable {
        emit LogA(address(this));
    }

    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }

    function deactivate(address to) public {
        selfdestruct(payable(to));
    }
}

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

6. Test case for “DELEGATECALL does not propagate msg.value to the implementation contract”

When calling `A.setVars{value: 5}(0xB, 4)`, the `value` param is not set, while `num` and `sender` are set correctly. However, the code below works as expected on the Goerli testnet.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

// NOTE: Deploy this contract first
contract B {
    // NOTE: storage layout must be the same as contract A
    uint public num;
    address public sender;
    uint public value;

    function setVars(uint _num) public payable {
        num = _num;
        sender = msg.sender;
        value = msg.value;
    }
}

contract A {
    uint public num;
    address public sender;
    uint public value;

    function setVars(address _contract, uint _num) public payable {
        // A's storage is set, B is not modified.
        (bool success, bytes memory data) = _contract.delegatecall(
            abi.encodeWithSignature("setVars(uint256)", _num)
        );
    }
}
```

7. Test case for “FEVM call instruction computational overhead differs slightly from standard EVM”

The following example Solidity contract defines a recursive function in a contract and benchmarks its execution with a different number of recursive calls.

In the EVM, there is an increment of 58% when doubling the number of recursive calls, while there is an increment of 91% in the FEVM.

```
contract Recursive {
    event Received(uint256 x, bool success);

    function foo(uint256 x) public payable {
        if (x > 0) {
            (bool success, bytes memory data) = address(this).call{
                value: msg.value
            }(abi.encodeWithSignature("foo(uint256)", x - 1));

            emit Received(x, success);
        }
    }
}
```

	x = 10	x = 20
Ethereum EVM	52,580	83,570
FEVM	16,285,395	31,244,789