Security Audit Report for EOS EVM

Date: Sep 12, 2023

Version: 1.1

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2023.09.12
16:01:22 +08'00'

Contents

Report Manifest

Item	Description
Client	EOS Network Foundation
Target	EOS EVM

Version History

Version	Date	Description
1.0	July 28, 2023	First Version
1.1	Sep 12, 2023	Update commit hashes of fixes

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	C++
Approach	Semi-automatic and manual verification

The target of this audit is to review the implementation of the EOS EVM, which is a compatibility layer deployed on top of the EOS blockchain. The EOS EVM serves as an implementation of the Ethereum Virtual Machine (EVM). It is implemented in C++ and compiled to a WASM binary to be executed within the EOS blockchain. The EOS EVM utilizes a modified version of Silkworm and Evmone for the execution of the EVM operations.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
EOS EVM[1]	`Version 1`	`8f649397c2f20cd66dd936440c735569bbf969c3`
	`Version 2`	`fd6d03b0d7aa63cc54d6d654601de31e126bb6e5`
Silkworm[2]	`Version 1`	`b8470c70901a2896fe85fc537b375f7b87a15923`
Evmone[3]	`Version 1`	`24365b7d5153972afb06d5ff0c51f5fecff19481`
Ethash[4]	`Version 1`	`8721c04c79584806c4f47f69684e01df6792cd48`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always

[1] https://github.com/eosnetworkfoundation/eos-evm/tree/main/contract

[2] https://github.com/eosnetworkfoundation/silkworm. Note that in this repository, only the source files related with EOS EVM contract are within the audit scope, while other files are out of scope.

[3] https://github.com/eosnetworkfoundation/evmone/tree/master/lib

[4] https://github.com/eosnetworkfoundation/ethash/tree/master/lib

recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the C++ language), the underlying compiling toolchain (e.g. the EOS blockchain and the CDT SDK) and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [5] and Common Weakness Enumeration [6]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[5] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[6] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **two** potential security issues. Besides, we also have **two** notes.

- High Risk: 2
- Note: 2

ID	Severity	Description	Category	Status
1	High	Lack of valid `ChainID` check	Software Security	Fixed
2	High	Potential incorrect state for smart contract destruction	Software Security	Fixed
3	-	Nonce issue of the reserved addresses	Note	-
4	-	The gas fee payment mechanism	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Lack of valid `ChainID` check

Severity High

Status Fixed in `Version 2` in the EOS EVM repository

Introduced by `Version 1` in the Silkworm repository

Description According to EIP-155, a transaction must have a (matched) `ChainID` for replay protection. However, in the transaction pre-validation, Silkworm only checks the `ChainID` when there is one. In this case, legacy (pre EIP-155) transactions can pass the pre-validation phase.

```
33  ValidationResult pre_validate_transaction(const Transaction& txn, uint64_t block_number, const
        ChainConfig& config,
34      const std::optional<intx::uint256>& base_fee_per_gas) {
35      const evmc_revision rev{config.revision(block_number)};
36
37      if (txn.chain_id.has_value()) {
38          if (rev < EVMC_SPURIOUS_DRAGON || txn.chain_id.value() != config.chain_id) {
39              return ValidationResult::kWrongChainId;
40          }
41      }
```

Listing 2.1: silkworm/core/silkworm/consensus/engine.cpp

Impact The validity of the `ChainID` is not checked if the transaction is of legacy type. It results in that legacy transactions of other chains can be replayed in the EOS EVM blockchain.

Suggestion Revise the `ChainID` check logic.

2.1.2 Potential incorrect state for smart contract destruction

Severity High

Status Fixed in `Version 2` in the EOS EVM repository

Introduced by `Version 1` in the EOS EVM repository

Description EOS EVM uses several `multi_index` tables to store EVM account and storage state on the EOS blockchain:

- `account_table` is used to store the account information (nonce, balance and code hash).
- `storage_table` is used to store the key-value pairs for the smart contracts.
- `gc_store_table` is used to store the deleted accounts.

Specifically, when a smart contract is self-destructed, the account ID of the smart contract would be recorded into the `gc_store_table`. The `gc` action would iterate over the `gc_store_table` and perform the real deletion on the `storage_table`.

When an account in the EOS EVM becomes active through the creation of a smart contract, a new incremental ID (aka a *primary key*) is generated in the `account_table`. This same ID is then used to initialize the entry for the smart contract in the `storage_table`. However, when a smart contract is destroyed, the corresponding account entry in the `account_table` is removed, and the account ID for the destroyed contract is logged into the `gc_store_table`, awaiting removal by the `gc` action.

However, the primary key for a `multi_index` table is not monotonically increasing. The removal of the entry in the `account_table` can result in the decrease of the primary key in a `multi_index` table. Besides, the deletion of the destructed contract in the `account_table` (immediately) and `storage_table` (after the `gc` action) is **asynchronous**, which can result in the incorrect state of the created contracts.

For example, a newly created smart contract may end up with an incorrect state if the following conditions are met within the same transaction:

1. The transaction creates a new contract, referred to as contract `A`.
2. It simultaneously destroys an existing contract, referred to as contract `B`.
3. Contract `B`, which is being destroyed, has the largest primary key.

If all the above conditions are met, contract `A` could potentially *inherit* all the storage slots from contract `B`. Moreover, if a `gc` action is later invoked, the storage slots of contract `A` might also be cleared.

The following code snippet illustrates the logic for updating the `account_table`. The `emplace` closure is used for creating a new state entry and invokes the `available_primary_key` method to generate a new primary key.

```
95    void state::update_account(const evmc::address& address, std::optional<Account> initial,
96                               std::optional<Account> current) {
97        check(!_read_only, "ro state");
98        const bool equal{current == initial};
99        if(equal) return;
100
101       account_table accounts(_self, _self.value);
102       auto inx = accounts.get_index<"by.address"_n>();
103       auto itr = inx.find(make_key(address));
104       ++stats.account.read;
105
106       auto emplace = [&](auto& row) {
107           row.id = accounts.available_primary_key();
108           row.eth_address = to_bytes(address);
109           row.nonce = current->nonce;
110           row.balance = to_bytes(current->balance);
111           // Codes are not supposed to changed in this call.
112           row.code_id = std::nullopt;
```

```
113         };
```

Listing 2.2: contract/src/state.cpp

The following code snippet [1] shows the logic for the `available_primary_key` method of the `multi_index` table. This function determines the next primary key by finding the largest key (i.e., the end of the iterator). Therefore, if the entry of the largest key is deleted from a `multi_index` table, the `available_primary_key` would be decreased.

```
1351    uint64_t available_primary_key()const {
1352       if( _next_primary_key == unset_next_primary_key ) {
1353          // This is the first time available_primary_key() is called for this multi_index
                instance.
1354          if( begin() == end() ) { // empty table
1355             _next_primary_key = 0;
1356          } else {
1357             auto itr = --end(); // last row of table sorted by primary key
1358             auto pk = itr->primary_key(); // largest primary key currently in table
1359             if( pk >= no_available_primary_key ) // Reserve the tags
1360                _next_primary_key = no_available_primary_key;
1361             else
1362                _next_primary_key = pk + 1;
1363          }
1364       }
1365
1366       eosio::check( _next_primary_key < no_available_primary_key, "next primary key in table is
              at autoincrement limit");
1367       return _next_primary_key;
1368    }
```

Listing 2.3: cdt/3.1.0/include/eosiolib/contracts/eosio/multi_index.hpp

Impact A newly created contract may inherit all the state from a previously destructed contract if certain conditions are met.

Suggestion Revise the smart contract destruction logic.

2.2 Notes

2.2.1 Nonce issue of the reserved addresses

Introduced by `Version 1` in the EOS EVM repository

Description The EOS EVM contract includes a feature for bridging between the EOS blockchain and EOS EVM, which involves creating special "reserved addresses". During the execution of the `execute_tx` function, the nonce and balance of the reserved address are overwritten. It's worth noting that this code snippet occurs before transaction validation.

```
227    if(from_self) {
228       check(is_reserved_address(*tx.from), "actions from self without a reserved from address are
             unexpected");
```

[1] The code snippet is part of the CDT SDK: https://github.com/AntelopeIO/cdt. This repository is not within the scope of this audit.

```
229       const name ingress_account(*extract_reserved_address(*tx.from));
230
231       const intx::uint512 max_gas_cost = intx::uint256(tx.gas_limit) * tx.max_fee_per_gas;
232       check(max_gas_cost + tx.value < std::numeric_limits<intx::uint256>::max(), "too much gas");
233       const intx::uint256 value_with_max_gas = tx.value + (intx::uint256)max_gas_cost;
234
235       populate_bridge_accessors();
236       balance_table.modify(balance_table.get(ingress_account.value), eosio::same_payer, [&](
              balance& b){
237          b.balance -= value_with_max_gas;
238       });
239       inevm->set(inevm->get() += value_with_max_gas, eosio::same_payer);
240
241       ep.state().set_balance(*tx.from, value_with_max_gas);
242       ep.state().set_nonce(*tx.from, tx.nonce);
243    }
```

Listing 2.4: contract/src/actions.cpp

Transactions with reserved addresses as the `from` address have no limitations, which means that these addresses can invoke and create smart contracts. In fact, a reserved address can create smart contracts of different codes on the same address by using the `SELFDESTRUCT` opcode.

However, there is currently an access control in place that requires the `execute_tx` function to be called by the contract itself if the `from` address of the transaction is a reserved address. The EOS EVM contract only calls itself in order to bridge token transfers between the EOS blockchain and EOS EVM. It's important to note that any future updates to the EOS EVM contract must not allow reserved addresses to create smart contracts, as this would violate the account design in EVM and could lead to potential problems.

2.2.2 The gas fee payment mechanism

Introduced by `Version 1` in the EOS EVM repository

Description The `pushtx` action is the entry function in the EOS EVM contract, which allows users to send their EVM transactions to the contract. However, there is an additional `miner` parameter in this function that specifies which account will receive the transaction fees. Initially, before the execution of the entire transaction, the fee recipient is set to the EOS EVM contract itself. However, after the transaction execution, a portion of the gas fees are transferred to the account specified by the `miner`.

```
374 void evm_contract::pushtx( eosio::name miner, const bytes& rlptx ) {
375    LOGTIME("EVM START");
376
377    assert_unfrozen();
378
379    eosio::check((get_sender() != get_self()) || (miner == get_self()),
380                 "unexpected error: EVM contract generated inline pushtx without setting itself as
                         the miner");
```

Listing 2.5: contract/src/actions.cpp

The current design of the EOS EVM contract has a potential front-running problem, which means that malicious actors can front-run any transaction by calling the `pushtx` action and replacing the `miner`

parameter with their own address. Through front-running, these actors can make risk-free profits if the gas fees paid by the transaction are greater than the CPU and NET fees of the EOS transaction.

Besides, according to the contract specifications, the EOS EVM contract itself is responsible for paying the fees associated with state storage. The relevant code snippet in the `pushtx` function is shown below.

```
394    evm_runtime::state state{get_self(), get_self()}; // @audit: the second parameter ram_payer =
           get_self()
395    silkworm::ExecutionProcessor ep{block, engine, state, *found_chain_config->second};
```

Listing 2.6: contract/src/actions.cpp

If the RAM fee required for writing a slot in the EVM is greater than the gas fee paid by the transaction sender, it could potentially lead to a DoS situation. In such a scenario, user activity would gradually decrease the balance of the EOS EVM contract, eventually leading to all user transactions failing due to insufficient funds to pay for RAM fees.