Telos EVM Audit Part 2

Prepared on: 2024-05-10

Contract: AUD522

Prepared by:
charles Holtzkampf

Prepared for:
Telos

Table of Contents

3 Executive Summary

4 Severity Description

5 Methodology

5 MEDIUM

Executive Summary

This document outlines any issues found during the audit of the contracts:

- telos.evm
- 8340e250923633676bef5ef568c65dc63822e2c1

- The contract has 0 remarks.
- 0 Minor security issues were found.
- 0 Major security issues were found.
- 0 Critical security issues were found.
- The risk associated with this contract is low

REMARK	MINOR	MAJOR	CRITICAL
0	0	0	0

Severity Description

REMARK

Remarks are instances in the code that are worthy of attention, but in no way represent a security flaw in the code. These issues might cause problems with the user experience, confusion with new developers working on the project, or other inconveniences.
Things that would fall under remarks would include:

- Instances where best practices are not followed
- Spelling and grammar mistakes
- Inconsistencies in the code styling and structure

MINOR

Issues of Minor severity can cause problems in the code, but would not cause the code to crash unexpectedly or for funds to be lost. It might cause results that would be unexpected by users, or minor disruptions in operations. Minor problems are prone to become major problems if not addressed appropriately.
Things that would fall under minor would include:

- Logic flaws (excluding those that cause crashes or loss of funds)
- Code duplication
- Ambiguous code

MAJOR

Issues of major security can cause the code to crash unexpectedly, or lead to deadlock situations.
Things that would fall under major would include:

- Logic flaws that cause crashes
- Timeout exceptions
- Incorrect ABI file generation
- Unrestricted resource usage (for example, users can lock all RAM on contract)

CRITICAL

Critical issues cause a loss of funds or severely impact contract usage.
Things that would fall under critical would include:

- Missing checks for authorization
- Logic flaws that cause loss of funds
- Logic flaws that impact economics of system
- All known exploits (for example, on_notification fake transfer exploit)

Methodology

Throughout the review process, we check that the token contract:

- Documentation and code comments match logic and behaviour
- Is not affected by any known vulnerabilities

Our team follows best practices and industry-standard techniques to verify the proper implementation of the smart contract. Our smart contract developers reviewed the contract line by line, documenting any issues as they were discovered. Ontop of the line by line review, we also perform code fuzzzing.

Our strategies consist largely of manual collaboration between multiple team members at each stage of the review, including:

I. Due diligence in assessing the overall code quality of the codebase.
II. Testing contract logic against common and uncommon attack vectors.
III. Thorough, manual review of the codebase, line-by-line.

Our testing includes:

- Overflow Audit
- Authority Control
- Authority Vulnerability
- Re-entry
- Timeout
- RAM Attacks
- Fake contract
- Fake deposit
- Denial of Service
- Design Logic Audit
- RNG attacks
- Stack Injection attacks

Our Code Fuzzing methodology:

We use a modified **EOSIO Contract Development Kit (CDT)** and custom harnessing to make EOS contracts suitable for fuzzing. The contract is instantiated and it's public functions are called in random order, with random input, so as to explore the state space and find corner case inputs that might lead to undesired outcomes. Apart from detecting logic bugs, this approach allows us also to detect memory bugs, hangs, undefined behavior and crash bugs in a semi-automated manner. Since EOS contracts are usually designed to run and complete within a short amount of time, fuzzing them is very fast as well, and therefore an effective instrument for teasing out bugs within the duration of an audit.

MEDIUM

1 Precompiles lack return value checks

Some precompiles can have an internal failure, for example if the point provided to the bn256 precompiles is invalid. In that case the error callback should be run instead of the sucess callback.

Emperically I've found that checking the return value of eosio::alt_bn128_add:

```
telos.evm/eosio.evm/src/precompiles/bncurve.cpp
Line 32 in 9f2024a
 eosio::alt_bn128_add( (char*) p1_bytes.data(), p1_bytes.size(), (char*)
↪  p2_bytes.data(), p2_bytes.size(), (char*) result.data(), result.size());
and eosio::alt_bn128_mul:
```

```
telos.evm/eosio.evm/src/precompiles/bncurve.cpp
Line 60 in 9f2024a
 eosio::alt_bn128_mul( (char*) p_bytes.data(), p_bytes.size(), (char*)
↪  mul_bytes.data(), mul_bytes.size(), (char*) result.data(), result.size());
```

and doing

```
throw_error(Exception(...), {});
return;
```

if the return value from the precompile not return_code::success is sufficient to become equivalent with Geth, at least in my fuzzer, but I will do more exhaustive testing to see if other precompiles need this treatment as well.

precompile_bnpairing is also affected. You need to change:

```
telos.evm/eosio.evm/src/precompiles/bncurve.cpp
Line 94 in 9f2024a
 result[31] = eosio::alt_bn128_pair( (char*) input.data(), input.size()) == 0;
into something like:

        const int r = eosio::alt_bn128_pair( (char*) input.data(), input.size());
        if ( r < 0 ) {
            throw_error(Exception(ET::outOfGas, "Out of Gas"), {});
            return;
        }
        result[31] = r == 0;
```

For ecrecover you need to do something like this:

```
    if ( eosio::k1_recover(  (char*)recoverable_sig.data(), recoverable_sig.size(),
                            (char*)output.data(), output.size(),
                            (char*)sig.data(), sig.size()) < 0 ) {
        precompile_return({});
        return;
    }
```

to match Geth's behavior here: https://github.com/ethereum/go-ethereum/blob/304879da20200f6912d24

Input to ecrecover that you can use for testing this:

```
6b8d2c81b11b2d699528dde488dbdf2f94293d0d34c32e347f255fa4a6c1f0a9000000000000000000000000000000000
```

Status: RESOLVED

2 CALL opcode gas computation: subtract GP_NEW_ACCOUNT before 63/64

If value > 0 in call then GP_NEW_ACCOUNT is subtracted after the 63/64 calculation:

63/64 at line 1503:

```
telos.evm/eosio.evm/src/instructions.cpp
Lines 1503 to 1504 in 9f2024a
 const auto gas_allowed = ctx->gas_left - (ctx->gas_left / 64);
 auto gas_limit = (_gas_limit > gas_allowed) ? gas_allowed : _gas_limit;
```

Subtract GP_NEW_ACCOUNT at line 1515:

```
telos.evm/eosio.evm/src/instructions.cpp
Line 1515 in 9f2024a
 bool error = use_gas(GP_NEW_ACCOUNT);
```

This needs to be switched around because Geth does this too:

Subtract GP_NEW_ACCOUNT (and the other things) at https://github.com/ethereum/go-ethereum/blob/ae470044878f15beb67eb7e66c117c9ad48f3a7b/core/vm/gas_table.go#L374

Then call callGas: https://github.com/ethereum/go-ethereum/blob/ae470044878f15beb67eb7e66c117c9a

Which does the 63/64 calculation: https://github.com/ethereum/go-ethereum/blob/ae470044878f15beb6

Status: RESOLVED

3 Modexp null pointer dereference

```
telos.evm/eosio.evm/src/precompiles/expmod.cpp
Line 43 in 9f2024a
 e = intx::be::unsafe::load<uint256_t>(read_input_vec(input,
 ↪  static_cast<uint64_t>(bl), static_cast<uint64_t>(el)).data());
```

If `static_cast<uint64_t>(el) == 0`, then `read_input_vec` will return an empty vector. An empty vector, when dereferenced (e.g. call the `.data()` method), returns NULL. `intx::be::unsafe::load<uint256` will then dereference this and cause a null pointer dereference.

It might also be possible that if `static_cast<uint64_t>(el)` is in the range 1..31, an out of bounds read will occur, or other unintended behavior, because `intx::be::unsafe::load<uint256_t>` will always assume an input of 32 bytes.

Essentially, to fix this, read_input_vec will always need to return a 32 byte vector, zero-padding it if the size parameter is less than 32.

Status: RESOLVED

4 Modexp gas calculation overflow

```
telos.evm/eosio.evm/src/precompiles/expmod.cpp
Line 75 in 9f2024a
 uint256_t gas_cost_bn = blen > mlen ? blen : mlen;
```

A `uint256_t` is used for the temporary modexp gas calculation. If the initial value is sufficiently large, then an overflow can occur on subsequent lines. E.g. assume `gas_cost_bn` is initially `0xfffe000000000000000000000000000000000000000000000000000000000000`.

Then apply

```
telos.evm/eosio.evm/src/precompiles/expmod.cpp
Line 76 in 9f2024a
 gas_cost_bn = ( gas_cost_bn + 7 ) / 8;
gas_cost_bn is now 0x1fffc00000000000000000000000000000000000000000000000000000000000.
```

Then apply

```
telos.evm/eosio.evm/src/precompiles/expmod.cpp
Line 77 in 9f2024a
 gas_cost_bn *= gas_cost_bn;
```

`gas_cost_bn` is now 0, but it should be `0x3fff000100000000000000000000000000000000000000000000000`

This will make the operation essentially free while allocating a huge array later on in `read_input_vec`, which causes a crash.

Geth uses real bignums rather than constrained integer types like `uint256_t`. It might be possible to correctly compute the gas using `uint512_t` or `uint1024_t` but to avoid any uncertainty, it would be better to use an actual bignum library like Boost multiprecision if possible.

Status: RESOLVED

5 CREATE/CREATE2 does not check for nonce overflow

If the invoking address has a nonce which is 2**64-1, then CREATE/CREATE2 must fail: https://github.com/ethereum/go-ethereum/blob/304879da20200f6912d241ccd471e140d3487093/core/v
L435

Status: RESOLVED

6 CREATE/CREATE2 does not pass value

This can result in execution trace differences with Geth if the CREATE'd contract invokes CALLVALUE.

To fix, change:

```
telos.evm/eosio.evm/src/instructions.cpp
Line 1418 in 9f2024a
 0, // Value
```

to:

```
contract_value,
```

For reference, Geth passes the value to the newly created contract here: https://github.com/ethereum/go-ethereum/blob/767b00b0b514771a663f3362dd0310fc28d40c25/core/vm/evm.go#L461

Status: RESOLVED

7 Special case in SELFDESTRUCT not added to checkpoint

SELFDESTRUCT has special logic to handle the case contract_address == recipient_address:

```
telos.evm/eosio.evm/src/instructions.cpp
Lines 1951 to 1954 in 9f2024a
 auto accounts_byaddress = contract->_accounts.get_index<eosio::name("byaddress")>();
 accounts_byaddress.modify(accounts_byaddress.iterator_to(ctx->callee),
 ↪  eosio::same_payer, [&](auto& a) {
   a.balance = 0;
 });
```

The consequence of using custom logic instead of using `transfer_internal` is that the transfer is not added to the checkpoint, in case the transfers need to be reverted.

I think the solution is to change this whole block:

```
    if (contract_address == recipient_address)
    {
      auto accounts_byaddress =
↪  contract->_accounts.get_index<eosio::name("byaddress")>();
      accounts_byaddress.modify(accounts_byaddress.iterator_to(ctx->callee),
↪  eosio::same_payer, [&](auto& a) {
        a.balance = 0;
      });
    }
    // Transfer all balance
    else
    {
      bool transfer_error = transfer_internal(contract_address, recipient_address,
↪  balance);
      if (transfer_error) return;
    }
  }
```

to

```
        bool transfer_error = transfer_internal(contract_address, recipient_address,
↪  balance);
      if (transfer_error) return;
```

I've found this bug with the following set up. First a description of how it is supposed to go, then a note about what Telos does wrong:

Prestate: Account 0x000000000000000000000000636F6E7472616374 has balance 123 and code 5b5a586074600060039605f60026073f5605f60006019f535a058583830fff4187b20012a0a

Call 0x000000000000000000000000636F6E7472616374.

Step 1: At call depth 1, 0x000000000000000000000000636F6E7472616374 CREATE2's 0x2504595d475593ee020EbA31E14aA75AeD360476. transfer(from=0x00000000000000000000000063
to=0x2504595d475593ee020EbA31E14aA75AeD360476, value=115).

Step 2: At call depth 2, 0x2504595d475593ee020EbA31E14aA75AeD360476 CREATE2's 0xd6a19e3A4049dd5c33777E34e031e65d58B8279a. transfers(from=0x2504595d475593ee020EbA31E1
to=0xd6a19e3A4049dd5c33777E34e031e65d58B8279a, value=115)

Step 3: 0xd6a19e3A4049dd5c33777E34e031e65d58B8279a calls SELFDESTRUCT with beneficiary 0xd6a19e3A4049dd5c33777E34e031e65d58B8279a. set_balance(address=0xd6a19e3a4049dd5c3
value=0)

Step 4: 0x2504595d475593ee020EbA31E14aA75AeD360476 runs LOG0 which fails due to a stack underflow. Restores 0x000000000000000000000000636F6E7472616374 balance to 123.

In Step 4, upon reverting due to the LOG0 stack overflow, Telos tries to reverse the transfer in step 2 by sending 115 from 0xd6a19e3A4049dd5c33777E34e031e65d58B8279a back to 0x2504595d475593ee020EbA31E14aA75AeD360476. But 0xd6a19e3A4049dd5c33777E34e031e65d58B8 has 0 balance due to step 3, so this incorrectly fails.

Status: RESOLVED

8 Account constructor leaves nonce and index uninitialized

```
telos.evm/eosio.evm/src/account.cpp
Line 129 in 9f2024a
 auto [it, inserted] = empty_accounts_cache.try_emplace(address, address);
```

The second argument to try_emplaces invokes this Account constructor:

```
telos.evm/eosio.evm/include/eosio.evm/tables.hpp
Line 17 in 9f2024a
 Account (uint256_t _address): address(addressToChecksum160(_address)) {}
```

This only initializes the `address` field. Other fields in `Account` have default constructors initializing them, except `nonce` and `index`, which are native C++ types (`uint64_t`) which aren't initialized automatically.

It would be better to assign a default value to these in the class declaration:

```
telos.evm/eosio.evm/include/eosio.evm/tables.hpp
Line 9 in 9f2024a
 uint64_t index;
```

```
telos.evm/eosio.evm/include/eosio.evm/tables.hpp
Line 12 in 9f2024a
 uint64_t nonce;
```

Empirically I've found at least one instance where this can cause invalid executions: `call()` calls `get_account()`:

```
telos.evm/eosio.evm/src/instructions.cpp
Line 1455 in 9f2024a
 const Account& to_account = get_account(toAddress);
```

The returned account's nonce is later checked to be non-zero, depending on which additional gas is subtracted:

```
telos.evm/eosio.evm/src/instructions.cpp
Line 1514 in 9f2024a
 if (new_callee.is_empty()) {
```

Status: RESOLVED

9 Account constructor leaves nonce and index uninitialized

`Processor::create()` constructs the `init_code` variable. When `Processor::create()` returns to caller, `init_code` is deconstructed and freed. After that, the `success_cb` lambda function may be invoked. If the `PRINT_RECEIPT` preprocessor variable is defined, `success_cb` will then access and move `init_code`. This constitutes a use-after-free bug. Upon accessing `init_code` at this point, its contents may be anything, and the presence of this instance of undefined behavior may alter the control flow of the EVM altogether.

The issue is easily fully resolved by capturing init_code as part of the lambda expression. Change this line:

```
telos.evm/eosio.evm/src/instructions.cpp
Line 1223 in 9f2024a
 auto success_cb = [&, p_ctx, new_account_address, gas_limit, sm_checkpoint](const
↪  std::vector<uint8_t>& output, const uint256_t& sub_gas_used) {
```

to

```
 auto success_cb = [&, init_code, p_ctx, new_account_address, gas_limit,
↪  sm_checkpoint](const std::vector<uint8_t>& output, const uint256_t& sub_gas_used)
↪  {
```

Status: RESOLVED

10 Use-after-free in Processor::create() with PRINT_RECEIPT enabled

`Processor::create()` constructs the `init_code` variable. When `Processor::create()` returns to caller, `init_code` is deconstructed and freed. After that, the `success_cb` lambda function may be invoked. If the `PRINT_RECEIPT` preprocessor variable is defined, `success_cb` will then access and move `init_code`. This constitutes a use-after-free bug. Upon accessing `init_code` at this point, its contents may be anything, and the presence of this instance of undefined behavior may alter the control flow of the EVM altogether.

The issue is easily fully resolved by capturing init_code as part of the lambda expression. Change this line:

```
telos.evm/eosio.evm/src/instructions.cpp
Line 1223 in 9f2024a
 auto success_cb = [&, p_ctx, new_account_address, gas_limit, sm_checkpoint](const
↪  std::vector<uint8_t>& output, const uint256_t& sub_gas_used) {
```

to

```
 auto success_cb = [&, init_code, p_ctx, new_account_address, gas_limit,
↪  sm_checkpoint](const std::vector<uint8_t>& output, const uint256_t& sub_gas_used)
↪  {
```

Status: RESOLVED