Security Audit – Neon EVM

Lead Auditor: Robert Reith

Second Auditor: Sebastian Fritsch

Second Auditor: Mathias Scherer

Administrative Lead: Thomas Lambertz

April 16th 2024

Table of Contents

Executive Summary

Neodyme audited updates to **Neon'** on-chain EVM program during spring 2024. Due to the specific threat model of evm programs, the scope of this audit included implementation security, overall design and architecture. The auditors found that Neon's EVM program comprised a clean design and high code quality. According to Neodyme's Rating Classification, **1 critical and 1 high vulnerabilities** and **2 medium-severity issues** were found. The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

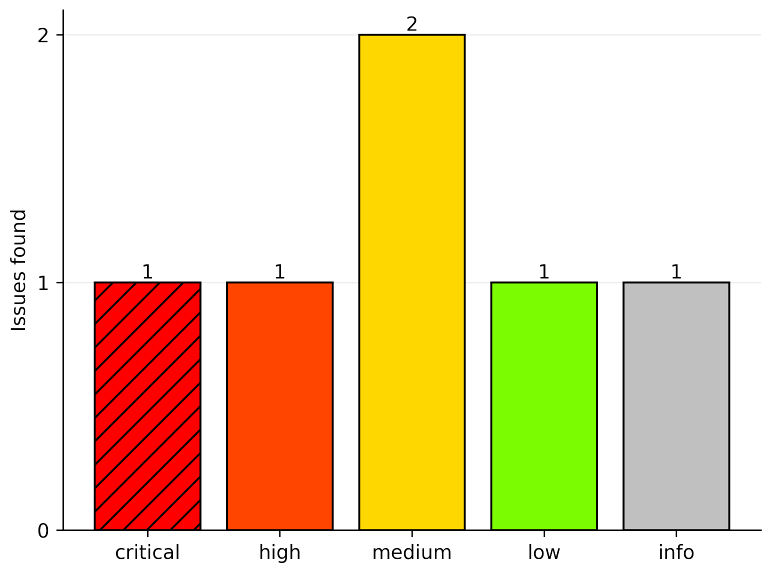


Figure 1: Overview of Findings

All findings were reported to the Neon developers and addressed promptly. The security fixes were verified for completeness by Neodyme. In addition to these findings, Neodyme delivered the Neon team a list of nit-picks and additional notes that are not part of this report.

1 | Introduction

During spring 2024, Neon commissioned Neodyme to conduct a detailed security analysis of multiple updates to Neon' on-chain EVM program.

Three senior auditors performed the audit between 1st of March and 1st of April. This report details all findings from this time span.

The audit mainly focused on the contract's technical security, but also considered its design and architecture. After the introduction, this report details the audit's Scope, gives a brief Overview of the Contract's Design, then goes on to document our Findings.

Neodyme would like to emphasize the high quality of Neon' work. Neon' team always responded quickly and **competent** to findings of any kind. Their **in-depth knowledge** about evm programs was apparent during all stages of the cooperation. Evidently, Neon invested significant effort and resources into their product's security. Their **code quality is above standard**, as the code is very well documented, naming schemes are clear, and the overall architecture of the program is **well thought out, clean and coherent**. The tests are **covering all important cases**. The contract's source code has no unnecessary dependencies, relying mainly on its own custom framework.

Findings Summary

During the audit, **5 security-relevant** and **1 informational** findings were identified. Neon remediated all of those findings before the protocol's launch.

In total, the audit revealed:

1 critical • **1** high-severity • **2** medium-severity • **1** low-severity • **1** informational

issues.

The highest severity finding addresses a vulnerability where a reentrancy issue allowed for state corruption. The second highest severity finding addresses another issue with reentrancy, invalidating `STATICCALL` properties. Both of these findings were remediated by removing the ability to reenter into Neon altogehter.

All findings are detailed in section Findings.

2 | Scope

The contract audit's scope comprised of two major components:

- **Implementation** security of the source code
- Security of the **overall design**

All of the source code, located at https://github.com/neonlabsorg/neon-evm, is in scope of this audit. However, third-party dependencies are not. As Neon only relies on widely used standard libraries, this does not seem problematic. Relevant source code revisions are:

- `73237d331bb3806100c2e09bfe8c958809eab320` • Start of the audit
- `0e13c3599d2a820829f3786dc0648e954398d8df` • Security Fixes Completed
- `0e13c3599d2a820829f3786dc0648e954398d8df` • Last Reviewed Revision

3 | Project Overview

This section briefly outlines the updates to Neon' EVM functionality, which were in scope for this audit. To get a full description of the project, please read our initial audit report. In total, 4 significant new features were added to Neon in this audit: Account revisions, balance accounts, custom gas tokens and solana_call.

Account Revisions

One of the major changes is the introduction of revisions for Neon accounts. This change enables stepped transactions to execute in parallel. In previous versions, the EVM locked accounts that were used by a stepped transaction to prevent other transactions from changing them between execution cycles. With the replacement of account blocking with account revisions, transactions can now always be executed and don't have to wait for previous transactions to finish that access the same accounts. Revisions are counters that are stored within the StorageAccounts and ContractAccounts, which get incremented for each transaction that changes a value within these accounts. This enables the EVM to detect changed accounts before continuing a transaction execution. When such a change is detected, the EVM resets and restarts the execution of the affected transaction. With this process, the program can ensure that transactions are executed with the latest state of all involved accounts. At the beginning of our audit, BalanceAccounts didn't keep track of their revision, shifting the responsibility to smart contract developers to keep track of the native token balances within their contracts. After raising an issue regarding this design, Neon implemented revisions for BalanceAccounts as well. In addition to that, Neon added revisions for arbitrary Solana accounts by using a hash over their owner, lamports, and data.

Balance Accounts

The update implements a whole new account type, the Balance Account. It is a derivation out of the former ether account, now splitting native token holding logic from contract logic, and also having a separate solana account structure holding this information. This change makes particular sense in combination with account revisions and custom gas tokens.

Custom Gas Tokens

The Neon team decided to add a feature that allows users to pay gas in different tokens. Previously, users of the Neon EVM had to pay for gas with the Neon token as an equivalent to the Eth token on the

Ethereum blockchain. With this new change, the fees can be paid in other tokens, depending on the smart contract being called. To implement this, Neon uses the `chain_id` field within transactions, which is normally used to differentiate transactions meant for different EVM chains. The original purpose of this field is that a transaction issued and signed for Neon would not be valid on Ethereum, because their `chain_id` differs. Because this field is quite large, a `u32`, Neon can allocate many `chain_id`s and designate a separate gas token for each of them. This update comes with multiple changes. Firstly, each contract has to define on which `chain_id` it is deployed, which also means in which gas token it holds its native funds. Second, each address can have multiple balance accounts, one for each `chain_id`, storing the account's balance for that particular gas token. Third, any native token transaction, including all different types of `call`, is updated to correctly handle `chain_id`, for instance rejecting transfers from one balance account to another if their `chain_id` doesn't match.

solana_call

The last change added in this update is the ability of smart contracts on Neon to call arbitrary solana programs. For this, a new precompile extension has been added which adds a new neon contract which in turn executes the native solana calls. It also provides some helpful functions to calculate solana addresses, PDAs, or crate Accounts.

4 | Findings

This section outlines all of our findings. They are classified into one of five severity levels, detailed in Appendix C. In addition to these findings, Neodyme delivered the Neon team a list of nit-picks and additional notes which are not part of this report.

All findings are listed in Table 2 and further described in the following sections.

Table 1: Findings

Name	Severity
[ND-NN2-C1] State corruption through reentrancy	Critical
[ND-NN2-H1] STATICCALL properties violated through reentrancy	High
[ND-NN2-M1] Balance Accounts Lack Account Revisions	Medium
[ND-NN2-M2] CALLCODE implementation does not adhere to specification	Medium
[ND-NN2-L1] SELFDESTRUCT does not conform to EIP-6780	Low
[ND-NN2-I1] Revision can overflow	Info

ND-NN2-C1 – State corruption through reentrancy

Severity	Impact	Affected Component	Status
Critical	State Corruption	Solana Call	Fixed

Description

Neon allows Solana compatibility through precompiles. One of these precompiles is the `SYSTEM_ACCOUNT_CALL_SOLANA` precompile, which allows an EVM contract to call Solana programs. If the EVM transaction is executed in a single Solana transaction, those Solana calls are executed right in place; in the case of a stepped transaction, those Solana calls are stored and executed after the EVM instruction has concluded.

Neon allows the external Solana instruction to call Neon again, which enables the following attack: 1. Create a Neon EVM instruction A, which modifies the storage of a contract (e.g. withdrawing on an ERC20 Contract) 2. Create a Neon EVM instruction B, which calls a contract with the following procedure: a) Use `SYSTEM_ACCOUNT_CALL_SOLANA` to call Neon EVM with instruction A. This appends an `ExternalInstruction` to the actions, which will be executed in the end. b) Modify the same contract storage as instruction A (e.g. withdrawing from the same ERC20 Contract but to a different address). This will queue an `EvmSetStorage` to be executed in the end. 3. Execute instruction B as a stepped instruction

After the transaction has concluded, `finalize` will call `apply_state_change`. It will execute the actions in the order they were appended. I.e. first `ExternalInstruction` will be executed and modify the contract's storage. Afterwards, `SetStorage` will be applied, and the contracts will be changed again, although they have already been changed. This allows, for example, the double spending of an ERC20 balance.

Location

- evm_loader/program/src/executor/precompile_extension/call_solana.rs#L297

Relevant Code

```
async fn execute_external_instruction<State: Database>(
    state: &mut State,
    context: &crate::evm::Context,
    instruction: Instruction,
    signer_seeds: Vec<Vec<u8>>,
    required_lamports: u64,
) -> Result<Vec<u8>> {
    #[cfg(not(target_os = "solana"))]
    log::info!("instruction: {:?}", instruction);

    let called_program = instruction.program_id;
    solana_program::program::set_return_data(&[]);

    for meta in &instruction.accounts {
        if meta.pubkey == state.operator() {
            return Err(Error::InvalidAccountForCall(state.operator()));
        }
    }
```

Mitigation Suggestion

In our opinion, the core of the bug lies in a missing account revision check after applying an `ExternalInstruction`, but our favoured fix would be disallowing reentrancy through an `ExternalInstruction` call, as this would eliminate many edge cases. This can be simply achieved by checking that the called `program_id` is not the neon program itself.

Remediation

In 34a27392d9e7bb935f0121c6b188605e17ac19d5 Neon added a check to disallow reentrancy within the `SYSTEM_ACCOUNT_CALL_SOLANA`.
This change prevents any Neon smart contract from reentering into the Neon EVM, by checking if the Neon EVM program id is present in any of the accounts passed to the CPI.

ND-NN2-H1 – STATICCALL properties violated through reentrancy

Severity	Impact	Affected Component	Status
High	State Corruption	STATICCALL	Fixed

Description

EIP-214 (https://eips.ethereum.org/EIPS/eip-214) introduced the STATICCALL opcode, which disallows any modifications to the global state during the call. This can be circumvented in NEON by using an `ExternalInstruction`, which calls Neon again.

Location

- /evm_loader/program/src/executor/precompile_extension/call_solana.rs#L31

Relevant Code

```rust
pub async fn call_solana<State: Database>(
    state: &mut State,
    address: &Address,
    input: &[u8],
    context: &crate::evm::Context,
    // Underscore indicates variable is unused / unchecked in this
       function
    _is_static: bool,
) -> Result<Vec<u8>> {
```

Mitigation Suggestion

Add a check for `is_static != true` in `call_solana`.

Remediation

Neon followed our suggestion and implemented a fix by adding a check fo `is_static` in pull request 34a27392d9e7bb935f0121c6b188605e17ac19d5.

ND-NN2-M1 – Balance Accounts Lack Account Revisions

Severity	Impact	Affected Component	Status
Medium	Unexpected Behaviour	Balance Accounts	Fixed

Description

We found that while contract- and storage-accounts use account revisions to prevent cross-contamination during optimistic execution, balance-accounts are do not have revisions. As we understand this was a conscious decision as in most cases changes to balances are accompanied by changes to state. However there are some theoretic cases where contracts use balance as state, and their execution integrity would be compromised in a significant way. To ensure integrity in all possible cases, we recommend also adding revisions to balance accounts.

Location

- /evm_loader/program/src/account/ether_balance.rs#L18-L30

Relevant Code

```
#[repr(C, packed)]
pub struct Header {
    pub address: Address,
    pub chain_id: u64,
    pub trx_count: u64,
    pub balance: U256,
}
impl AccountHeader for Header {
    const VERSION: u8 = 0;
}

pub struct BalanceAccount<'a> {
    account: AccountInfo<'a>,
}
```

Mitigation Suggestion

Add account revisions to balance accounts.

Remediation

Neon implemented revisions for balance accounts in commit bdaf787c7268dc5f39334846fc512816d5c29386. In addition to that, they separated the operator balance account for gas payments, so that changes to this account will not affect transaction execution. Furthermore, Neon added the condition that stepped transactions are only reset if accounts that have been changed have already been accessed by the transaction before the change has happened. This change was added in commit 0e13c3599d2a820829f3786dc0648e954398d8df.

ND-NN2-M2 – CALLCODE implementation does not adhere to specification

Severity	Impact	Affected Component	Status
Medium	Unexpected Behaviour	CALLCODE	Fixed

Description

CALLCODE is an EVM instruction similar to DELEGATECALL. It differs from DELEGATECALL in two key aspects. Firstly, CALLCODE allows the modification of the msg.value property for the to-be-called contract. Secondly, the msg.sender property will be set to the contract that calls via CALLCODE, while DELEGATECALL will keep the msg.sender as the EOA. This second part is missing in the current CALLCODE implementation and can lead to unexpected consequences for an contract using CALLCODE
.

Location

- /evm_loader/program/src/evm/opcode.rs#L1194-L1198

Relevant Code

```
    pub async fn opcode_callcode(&mut self, backend: &mut B) -> Result<
        Action> {
        let gas_limit = self.stack.pop_u256()?;
        let address = self.stack.pop_address()?;
        let value = self.stack.pop_u256()?;
        // [...]
        let chain_id = self.context.contract_chain_id;
        let context = Context {
            value,
            code_address: Some(address),
            ..self.context
        };
        // [...]

        self.fork(
            Reason::Call,
            chain_id,
            context,
            code,
            call_data,
            Some(gas_limit),
        );
        backend.snapshot();
        // [...]
        log_data(&[b"ENTER", b"CALLCODE", address.as_bytes()]);

        if backend.balance(self.context.caller, chain_id).await? < value {
            return Err(Error::InsufficientBalance(
                self.context.caller,
                chain_id,
                value,
            ));
        }

        self.opcode_call_precompile_impl(backend, &address).await
    }
```

Mitigation Suggestion

This bug can be easily fixed by setting `context.caller` to the address of the current contract in `opcode_callcode`.

Remediation

Neon followed our suggestion and implemented a fix by adding the suggested context change setting `context.caller` to the address of the current contract in `opcode_callcode`.

The fix is implemented in pull request 975be3d78d40822c747eca33ddf48c8d2706401b.

ND-NN2-L1 – SELFDESTRUCT does not conform to EIP-6780

Severity	Impact	Affected Component	Status
Low	Unexpected Behaviour	SELFDESTRUCT	Fixed

Description

EIP-6780 (https://eips.ethereum.org/EIPS/eip-6780) proposes changes to the behaviour of the `SELFDESTRUCT` opcode. Neon partially implements those changes but still deletes previously written data if `SELFDESTRUCT` was executed in a transaction that is not the same as the contract calling SELFDESTRUCT was created.

Furthermore, if `SELFDESTRUCT` is called in a transaction that is not the same as the contract calling SELFDESTRUCT was created and "if the target is the same as the contract calling SELFDESTRUCT that Ether will be burnt". This functionality is currently missing in Neon.

Location

- /evm_loader/program/src/executor/action.rs#L53-L83

Relevant Code

```rust
pub fn filter_selfdestruct(actions: Vec<Action>) -> Vec<Action> {
    // Find all the account addresses which are scheduled to
       EvmSelfDestruct
    let accounts_to_destroy: std::collections::HashSet<_> = actions
        .iter()
        .filter_map(|action| match action {
            Action::EvmSelfDestruct { address } => Some(*address),
            _ => None,
        })
        .collect();

    actions
        .into_iter()
        .filter(|action| {
            match action {
                // We always apply ExternalInstruction for Solana
                   accounts
                // and NeonTransfer + NeonWithdraw
                Action::ExternalInstruction { .. }
                | Action::Transfer { .. }
                | Action::Burn { .. } => true,
                // We remove EvmSetStorage|EvmIncrementNonce|EvmSetCode if account is scheduled for destroy
                Action::EvmSetStorage { address, .. }
                | Action::EvmSetCode { address, .. }
                | Action::EvmIncrementNonce { address, .. } => {
                    !accounts_to_destroy.contains(address)
                }
                // SelfDestruct is only aplied to contracts deployed in the current transaction
                Action::EvmSelfDestruct { .. } => false,
            }
        })
        .collect()
}
```

Mitigation Suggestion

We recommend implementing the EIP in full.

Remediation

The Neon team decided to completely remove the `SELFDESTRUCT` opcode and replacing it with `SENDALL` as proposed in EIP-4758. This fix has been proposed and later merged with pull request d80f39bd7c0315789686906fc41f352a28b3984b.

ND-NN2-I1 – Revision can overflow

Severity	Impact	Affected Component	Status
Info	State Corruption	Account Revisions	WIP

Description

This is a very theoretical attack and we don't think it is actually feasible or economical right now. In `increment_revision` Neon uses a `wrapping_add` to increment the revision, which is a `u32` integer. This means that after ~4.2 billion transactions, the revision will be the same again. An attacker could halt a stepped transaction at revision x, execute 2^{32} instructions in between, modify the state, and the halted transactions would continue on the corrupted state.

From a feasibility perspective, this currently seems impossible. Assuming ~4 simple neon instructions in a single TX and a (very high) inclusion rate of ~100 TX per block, one could reach ~1000 revision increments per second, which would need ~50 days of full-time spamming to overflow the revision. From a financial perspective, this would probably cost around 10 Million USD at current SOL pricing and wouldn't stay unnoticed by chain users. Furthermore, the attack could be easily disrupted by stepping the halted transaction further through a different operator.

Location

- /evm_loader/program/src/account/ether_storage.rs#L286-L295

Relevant Code

```rust
pub fn increment_revision(&mut self, rent: &Rent, db: &AccountsDB<'a>)
    -> Result<()> {
        if super::header_version(&self.account) < HeaderWithRevision::
            VERSION {
            self.header_upgrade(rent, db)?;
        }

        let mut header = super::header_mut::<HeaderWithRevision>(&self.
            account);
        header.revision = header.revision.wrapping_add(1);

        Ok(())
    }
```

Mitigation Suggestion

In general, we would prefer to use a `checking_add` instead of a `wrapping_add` here, or even better, replace the revision with a `u64`.

Remediation

The Neon team proposed a fix where stepped transaction execution would be time-constrained. At time of this writing, the fix is work in progress, and is scheduled to be added in summer 2024.

A | Methodology

Neodyme prides itself on not being a checklist auditor. We adapt our approach to each audit, investing considerable time into understanding the program up front, exploring its expected behaviour, edge cases, invariants, and ways in which the latter could be violated. We use our uniquely deep knowledge of Solana internals and our years-long experience in auditing Solana programs to even find bugs that others miss. We often extend our audit to cover off-chain components in order to see how users could be tricked or the contract affected by bugs in those components.

Nonetheless, we also have a list of common vulnerability classes, which we always exhaustively look for. We provide a sample of this list below.

- Rule out common classes of Solana contract vulnerabilities, such as:
  - Missing ownership checks
  - Missing signer checks
  - Signed invocation of unverified programs
  - Solana account confusions
  - Redeployment with cross-instance confusion
  - Missing freeze authority checks
  - Insufficient SPL account verification
  - Missing rent exemption assertion
  - Casting truncation
  - Arithmetic over- or underflows
  - Numerical precision errors
- Check for unsafe designs which might lead to common vulnerabilities being introduced in the future
- Check for any other, as-of-yet unknown classes of vulnerabilities arising from the structure of the Solana blockchain
- Ensure that the contract logic correctly implements the project specifications
- Examine the code in detail for contract-specific low-level vulnerabilities
- Rule out denial of service attacks
- Rule out economic attacks
- Check for instructions that allow front-running or sandwiching attacks
- Check for rug pull mechanisms or hidden backdoors

B | Vulnerability Severity Rating

Critical Vulnerabilities that will likely cause loss of funds. An attacker can trigger them with little or no preparation, or they are expected to happen accidentally. Effects are difficult to undo after they are detected.

High Bugs that can be used to set up loss of funds in a more limited capacity, or to render the contract unusable.

Medium Bugs that do not cause direct loss of funds but that may lead to other exploitable mechanisms, or that could be exploited to render the contract partially unusable.

Low Bugs that do not have a significant immediate impact and could be fixed easily after detection.

Info Bugs or inconsistencies that have little to no security impact.

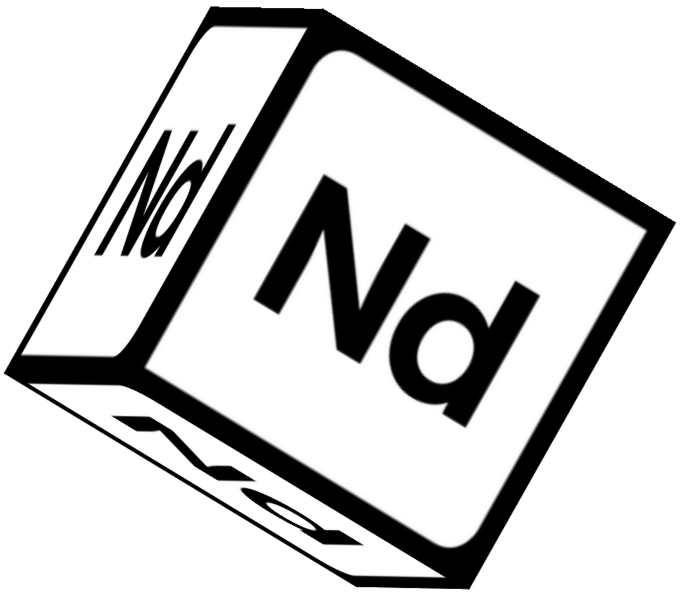
C | About Neodyme

Security is difficult.

To understand and break complex things, you need a certain type of people. People who thrive in complexity, who love to play around with code, and who don't stop exploring until they fully understand every aspect of it. That's us.

Our team never outsources audits. Having found over 80 High or Critical bugs in Solana's core code itself, we believe we have the most qualified auditors for Solana programs in our company. We've also found and disclosed critical vulnerabilities in many of Solana's top projects and have responsibly disclosed issues that could have resulted in the theft of over $10B in TVL on the Solana blockchain.

Our team met as participants in hacking competitions called CTFs. There, we competed and collaborated while finding vulnerabilities, breaking encryption, reverse engineering complicated algorithms, and much more. Through the years, many of our team members have won national and international hacking competitions and keep ranking highly among some of the hardest CTF events worldwide. In 2020, some of our members started experimenting with validators and became active members of the early Solana community. With the prospect of an interesting technical challenge and bug bounties, they quickly encouraged others from our CTF team to look for security issues in Solana. The result was so successful that after reporting several bugs, in 2021, the Solana Foundation contracted us for source code auditing. As a result, Neodyme was born.



Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
E-Mail: contact@neodyme.io

https://neodyme.io