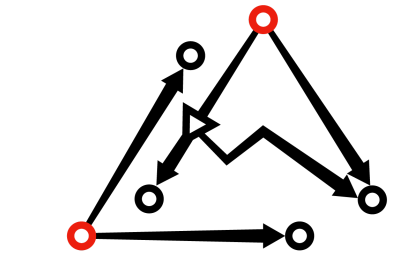

Least Authority

PRIVACY MATTERS

Neon EVM + Proxy Implementation

Security Audit Report

Neon Labs

Initial Audit Report: 17 November 2021

This Initial Audit Report is intended for internal use only. We advise against sharing this report beyond trusted team members and recommend that publication take place only after the verification has been completed and the Final Audit Report has been delivered.

Table of Contents

Overview

Background

Neon Labs requested that Least Authority perform a security audit of the Neon EVM, which includes the Rust EVM and EVM loader components, and the proxy implementation. Neon EVM is an Ethereum Virtual Machine (EVM) on Solana, which aims to enable Decentralized Application (dApp) developers to use Ethereum tooling to scale and get access to liquidity on Solana. This EVM is run by a Solana program called EVM Loader, which works around challenges introduced by the different designs of the two blockchains. The Neon proxy implementation serves as an interface between Solana RPC nodes and existing Ethereum frontends, allowing them to interact with the Neon EVM running inside Solana using the same API that is in use by native Ethereum by wrapping the calls in appropriate Solana API calls.

Project Dates

- **September 27 - November 11**: Code review (*Completed*)
- **November 17**: Delivery of Initial Audit Report (*Completed*)
- **TBD:** Verification completed
- **TBD:** Delivery of Final Audit Report

The dates for verification and delivery of the Final Audit Report will be determined upon notification from the Neon Labs team that the code is ready for verification.

Review Team

- ElHassan Wanas, Security Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Nicole Ernst, Security Researcher and Engineer
- Steven Jung, Security Researcher and Engineer
- Suyash Bagad, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Neon EVM and proxy implementation followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:

- Neon EVM
  - Rust EVM: https://github.com/neonlabsorg/evm
  - EVM Loader: https://github.com/neonlabsorg/neon-evm
- Proxy Implementation:
  - https://github.com/neonlabsorg/proxy-model.py/blob/develop/proxy/plugin/solana_rest_api.py
  - https://github.com/neonlabsorg/proxy-model.py/blob/develop/proxy/plugin/solana_rest_api_tools.py

Specifically, we examined the Git revisions for our initial review:

Rust EVM: `6076b6029b39d26e16180e910b49be3031950c5b`

EVM Loader: `81ed454e6ce3353c77eb910ccaa49eb5dde19c66`

Proxy Implementation: `b3b9d74d5cf47006823602f25cd7ed7884ac528b`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- Neon EVM
  - Rust EVM: https://github.com/LeastAuthority/NeonRustEVM
  - EVM Loader: https://github.com/LeastAuthority/NeonEVM
- Proxy Implementation: https://github.com/LeastAuthority/NeonProxy

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Neon Labs Documentation: https://doc.neonlabs.org/
- Neon EVM `README.md`: https://github.com/LeastAuthority/NeonRustEVM/blob/master/README.md
- Neon AMA - with Anatoly You Tube video: https://www.youtube.com/watch?v=qlSfvD--5IY#t=8m10s
- Whitepaper: https://solana.com/solana-whitepaper.pdf
- G. Wood, 2014, "Ethereum: A Secure Decentralised Generalised Transaction Ledger" [W14]
- Neodyme Audit Github Issues (*shared with Least Authority via Telegram on 20 October 2021*)
  - 258; 259; 261; 262; 263; 264; 265; 269; 278; 279; 280; 281; 295; 296; 297; 298; 299; 300; 301; 302; and 313.

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the smart contract;
- Attacks that impacts funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service attacks and security exploits that would impact or disrupt the execution of the smart contract;
- Vulnerabilities in the smart contract code;
- Proper management of encryption and signing keys;
- Protection against malicious attacks and other ways to exploit the smart contract;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Finding

General Comments

Neon EVM implements an EVM and its associated Solana-Ethereum state as a set of programs (i.e. smart contracts) in the Solana ecosystem.

Solana is a new and emerging layer-1 Distributed Ledger Technology (DLT) and it differs substantially from other blockchains. As a result, the attack surface has yet to be well established and the approach to Solana smart contract auditing, like any new ecosystem introducing new concepts and ideas, requires deviation from the approach taken with more mature and familiar ecosystems. Furthermore, the Solana ecosystem has not yet had the opportunity to mature and be adequately battle tested in production, which presents interesting challenges to security researchers and developers working with projects building on Solana. As a result, we recommend that the Neon Labs team continue to adhere to security best practices, monitor ongoing research and developments in the ecosystem as it relates to security improvements, and conduct regular security audits by independent security auditing teams.

Our team performed a broad and comprehensive review of the Neon EVM and the proxy implementation and found that the Neon Labs team has considered security in their system design and implementation. However, we identified several areas of improvement, as outlined in the issues and suggestions of this report, that would help to improve the code quality, dependency maintenance, and contribute to the overall security of the project. We recommend addressing the issues and suggestions in order to aid in preventing the possibility of more serious vulnerabilities entering the system.

Areas of Investigation

The Rust EVM is a customized fork of Parity's SputnikVM. We performed a comparison of the Rust EVM and SputnikVM implementations to check for adherence with the original implementation. The Neon Labs team customized the SputnikVM in two key areas. First, the Neon Labs team identified non-optimal code in the SputnikVM codebase and optimized it. To the best of our knowledge, the resulting optimizations add value to the SputnikVM code. As a result, the authors of the SputnikVM development team (i.e. Parity) might consider merging some of the Neon Labs team's optimizations upstream. The second area of customization are modifications that enable the EVM to function inside the Solana ecosystem.

We investigated the EVM loader for any issues concerning account permissions and account ownership and did not identify any ability to circumvent the implemented checks. In addition, we examined the usage of nonces to prevent transaction replay attacks and found no issues with the implementation of nonces.

We checked if accounts are blocked appropriately to prevent unintended behaviour due to access while another transaction is still running iteratively. We also checked if they are appropriately unblocked again to prevent deadlocks. In particular, we investigated the case where an operator starts a transaction and abandons it, and found that other operators have an economic incentive to pick up the abandoned transaction after the allotted waiting time has passed. We did not identify a way to disrupt the intended functionality of the system by operator abandonment of a transaction (See Account Blocking).

We examined the checks implemented against malformed transactions that could corrupt data and did not identify any attacks that would cause a malformed transaction to produce unintended behavior. We investigated the functions used for facilitating payment from operators to Neon EVM and did not identify any issues that could lead to operators dodging a payment or having to pay too much.

Finally, we performed comparative analysis of the Rust EVM implementation against the specification of Ethereum Virtual Machine [W14] and did not identify any deviation from the specifications. However, the Rust EVM does not implement recent EIP's, such as the Ethereum London hard fork (i.e. EIP-1559,

EIP-3198, EIP-3529, EIP-3541, and EIP-3554). This could make the Rust EVM implementation incompatible with some smart contracts as a result of introducing substantial changes. While the inclusion of recent EIPs is a design decision, we suggest that the Neon Labs team carefully evaluate the security implications of making any changes to the current implementation and, if introduced, to conduct follow up security audits.

System Design

The Solana ecosystem deviates significantly from the traditional blockchain paradigm and introduces many new and non-trivial concepts and interactions. In order to better understand the Neon EVM system design and review the system for security vulnerabilities, we analyzed the differences between the design of Ethereum and Solana and identified a number of challenges to providing an Ethereum-like environment within Solana, as detailed below.

Solana Statelessness

Ethereum is a state-rich blockchain, whereas programs on Solana are entirely state-less. For the Neon EVM, this means that the state of smart contracts has to be kept separate from the code in additional Solana accounts. These accounts must implement their own write access permissions and care must be taken that the data is not corrupted while being written to or read from these accounts.

Transaction Size

While the Ethereum network does not have an explicit limit on the size of transactions, a varying implicit size limit of several hundred kilobytes exists due to gas constraints. In the Solana network, the transaction size limit is the Maximum Transmission Unit (MTU) of an IPv6 network, minus additional header data, thus leaving a comparatively small 1,232 bytes. The Neon EVM seeks to work around this limitation in the Solana network by first writing large transactions into a holder account in Solana and then executing them iteratively. This holder account is vulnerable to and must be protected from malicious write attempts. In addition, transactions contained within the Solana holder account are vulnerable to and must be protected from replay attacks.

Computation Limitations

The Ethereum network uses a block gas limit to place constraints on the amount of computations available to single transactions, with one smart contract (in theory) being able to use up the entire gas available in a block, given that the caller sets the gas price at a sufficiently high-level. In contrast, the Solana network assigns a fixed computation budget to each instruction contained in a transaction that cannot be exceeded. To enable their users to send transactions that require extensive computation, an emulation functionality has been implemented in the Neon proxy implementation that tests how many steps of a transaction can be executed, and then proceeds to only send chunks of the transaction that will be executed. Operators are economically incentivized to keep executing such a transaction by being required to make a deposit to the Neon EVM at the start of such an iterative transaction, that is only returned once the entire transaction has been processed.

Parallel Transaction Execution

Ethereum processes each transaction iteratively, with miners deciding transaction order based on logic that maximizes miner profit. Solana can process as many transactions in parallel as there are threads available to the current validator, which has the practical effect of transactions being processed in a First-In, First-Out (FIFO) order. As a result, every Solana transaction must contain an array of accounts that the transaction seeks to access to prevent data races from concurrent access of the same account. To generate such arrays for Ethereum native transactions and enable their execution on the Solana network, the Neon Labs team utilizes the aforementioned emulation functionality to determine which accounts are accessed by a transaction, and to then form an array out of the account public keys for use in the eventual

Solana transaction. This introduces additional execution logic that may turn out to be an attack surface in the future. However, given Solana's relative novelty and the number of unknowns, this functionality has yet to be battle tested and it is difficult to reason about the potential security implications.

Account Blocking

The interplay with the iterative execution of Ethereum transactions is another way in which the Neon EVM is distinct, as all of the accounts in the aforementioned array have to be blocked for the duration of the multiple slots spanning transaction, which can lead to inconvenience for the users whose accounts are blocked (see Computation Limitations).

Code Quality

The Rust EVM repository is well organized and generally adheres to Rust development best practices. However, there is an instance of an implementation error in the function `is_valid`, which prevents the function from behaving as intended, for which we recommend implementing the suggested mitigation (Issue A). We also found that the proxy implementation repository was generally well written.

We identified several issues in the quality of the EVM loader implementation and found that it does not adhere to development best practices. For example, we found several unresolved TODOs in the codebase, which may lead to confusion about the code's completeness and intended functionality. As a result, we recommend resolving all TODOs (Suggestion 6). In addition, there is a lack of adherence to the latest dependency standards identified by the Rust advisory database (See Use of Dependencies).

In all three in-scope repositories, there are many instances of unused code in the implementation. Unused code creates confusion about the state of completeness of the codebase, as well as about the intended functionality of the system. We recommend that unused code be removed from the codebase (Suggestion 3). In addition, there is inconsistency in the naming convention for functions and variables, which may lead to confusion about intended behavior and functionality. This may result in unintended errors or security vulnerabilities being overlooked by maintainers or security researchers of the code. We recommend implementing a consistent variable and function naming convention and that names accurately describe the intended behavior of the variable or function (Suggestion 4).

Tests

The Rust EVM and proxy implementation repositories contain sufficient test coverage.

However, the EVM loader repository test suite does not sufficiently cover the codebase. The existing tests contain compilation errors and there are many areas within the EVM loader repository that contain no test coverage. As a result, we recommend that a comprehensive test suite be implemented that tests for all success, failure, and edge cases that could lead to unintended behavior resulting in errors or vulnerabilities (Suggestion 9).

Documentation

The project documentation was accurate and helpful in describing the entire Neon EVM system at a high-level. However, there is minimal documentation specific to the coded implementation. Since the Neon EVM implementation aims to realize non-trivial system design objectives, we recommend improving the documentation to better describe the logic and rationale for the deviations from the SputnikVM fork, the design objectives of these deviations, and how these objectives are achieved in the implementation (Suggestion 5).

Code Comments

The Neon EVM and the proxy implementation repositories would benefit from additional code comments describing the code's intended behavior. Given the nuanced functionality that is expected from each of the components of the system and the complexity of their interactions, we recommend that comprehensive code comments be implemented that provide detailed descriptions of the intended behavior of each function and component, which facilitates reasoning about the security properties of the system (Suggestion 2).

Scope

The in-scope repositories were sufficient and included all the security critical components of the system.

However, as previously noted, Solana is a new type of layer-1 DLT with a concurrent execution model. As a result, comprehensive threat models have not emerged and security research and tactics have yet to be appropriately established and battle tested. Thus, we recommend ongoing security audits by independent security auditing teams within the context of a well defined security roadmap. In addition, the Neon Labs team should continue to monitor security developments, such as new exploits and vulnerabilities, in order to mitigate any potential similar issues in the implementation of Neon EVM and the proxy implementation.

Use of Dependencies

We identified several issues resulting from the EVM loader implementation's lack of adherence to the latest dependency standards identified by the Rust advisory database. As a result, we recommend that the Neon Labs team address these vulnerabilities by using up-to-date dependencies that include the latest security patches, which is an effective protection measure against known vulnerabilities (Issue B; Issue C; Issue D; Issue E; Issue F; Suggestion 1).

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Ambiguity in the Function is_valid	Reported
Issue B: Data Race in crossbeam-deque	Reported
Issue C: Out-of-Bounds Write in nix::unistd::getgrouplist	Reported
Issue D: no-op in zeroize_derive v1.1 for ENUMs	Reported
Issue E: Symlinks Can Create Arbitrary Directories	Reported
Issue F: Failure is Officially Deprecated and Unmaintained	Reported
Suggestion 1: Consider Using socket2 Instead of net2	Reported
Suggestion 2: Improve and Increase Code Comments	Reported
Suggestion 3: Remove Unused Code	Reported

Suggestion 4: Improve Variable and Function Naming	Reported
Suggestion 5: Improve Documentation	Reported
Suggestion 7: Update Solidity Compiler Version	Reported
Suggestion 8: Use Explanatory Comments Instead of Declaring Unused Interface	Reported
Suggestion 9: Increase Test Coverage	Reported

Issue A: Ambiguity in the Function `is_valid`

Location

`NeonRustEVM/core/src/valids.rs`

Synopsis

The `is_valid` function in the Rust EVM must return `true` if a given position is a valid jump destination in a program. Given an input `position`, from L27 to L33 in the `is_valid` function, the data byte and the bit to be checked is computed as:

```
let byte_index = position / 8;
let byte = self.data[byte_index];

let bit_index = position % 8;
let bit_test = 1_u8 >> bit_index;

(byte & bit_test) == bit_test
```

The variable `bit_test` would be 0 if `bit_index > 0`. The function returns a boolean evaluated on L33. When `bit_index > 0`, the function returns `true`. The only case when this function returns `false` is when the variable `byte` is an even number and `position` is a multiple of 8, which is unintended behavior for the function.

Impact

While executing instructions in the Rust EVM, in functions such as `jump` and `jumpi`, the Rust EVM would rarely generate errors for invalid jumps. This can lead an attacker to tamper with the execution of certain EVM instructions and potentially lead the program to malfunction. The attacker can carefully modify the instructions or inputs to prevent this function from claiming an invalid jump.

Mitigation

We recommend the expression on L31 be corrected to:

```
let bit_test = 1_u8 << bit_index;
```

In other words, the bit to be tested must be computed by the left shift of `1_u8` instead of right shift.

Status

Reported.

Issue B: Data Race in crossbeam-deque

Location

`NeonEVM/evm_loader/Cargo.lock`

`https://rustsec.org/advisories/RUSTSEC-2021-0093.html`

Synopsis

The EVM loader repository uses the `crossbeam-deque` crate in version 0.7.3 and 0.8.0. However, according to the Rust Advisory Database, those particular versions have a memory-corruption vulnerability and are marked as yanked.

Impact

The Rust Advisory Database notes, "In the affected versions of this crate, the result of the race condition is that one or more tasks in the worker queue can be popped twice instead of other tasks that are forgotten and never popped. If tasks are allocated on the heap, this can cause double free and a memory leak. If not, this still can cause a logical bug."

Remediation

We recommend an upgrade to one of the following versions: >=0.7.4, <0.8.0 or >=0.8.1.

Status

Reported.

Issue C: Out-of-Bounds Write in nix::unistd::getgrouplist

Location

`NeonEVM/evm_loader/Cargo.lock`

`https://rustsec.org/advisories/RUSTSEC-2021-0119.html`

Synopsis

The EVM loader repository uses the `nix` crate in version 0.19.1 and 0.22.1. However, according to the Rust Advisory Database, those particular versions have a memory-corruption vulnerability.

Impact

The GitHub issue outlining the vulnerability states, "If `libc::getgrouplist` returns -1, indicating the supplied groups buffer is too short to hold all the user's groups, the current code will double the buffer and try again. Unfortunately, the `ngroups` value it passes to `libc::getgrouplist` does not reflect the length of the buffer. After the 1st iteration in this scenario, `libc` will set `ngroups` to the number of groups it found, which can be a larger number than the doubling of the `group`'s capacity. The 2nd iteration of the loop will now have a mismatch between the `group`'s capacity and the `ngroups` value it supplies. This will lead to `libc` writing into unallocated memory beyond the buffer's allocation, leading to a segfault, allocator corruption (e.g. `free` on destroying the `Vec<Gid>` complaining of invalid size passed in) or more generally, UB behavior."

Remediation

We recommend an upgrade to a version ^0.20.2 OR ^0.21.2 OR ^0.22.2 or >=0.23.0.

Status

Reported.

Issue D: no-op in zeroize_derive v1.1 for ENUMs

Location

`NeonEVM/evm_loader/Cargo.lock`

`https://rustsec.org/advisories/RUSTSEC-2021-0115.html`

Synopsis

The EVM loader repository uses the `zeroize_derive` crate in version 1.1.0. However, according to the Rust Advisory Database, that particular version has a no-op bug and is marked as yanked.

Impact

The Rust Advisory Database states, "`#[zeroize(drop)]` does not generate a drop implement `Drop` for enums. This can result in memory not being zeroed out after dropping it, which is exactly what is intended when adding this attribute."

Remediation

We recommend an upgrade to a version >=1.1.1.

Status

Reported.

Issue E: Symlinks Can Create Arbitrary Directories

Location

`NeonEVM/evm_loader/Cargo.lock`

`https://nvd.nist.gov/vuln/detail/CVE-2021-38511`

Synopsis

The EVM loader repository uses the `tar` crate in version 0.4.35. However, according to the National Vulnerability Database, that particular version has vulnerability with a cvss score of 7.5 (high).

Impact

In the vulnerable versions, exploits can be written to write folders outside of the application's memory.

Remediation

We recommend an upgrade to a version >=0.4.36.

Status

Reported.

Issue F: Failure is Officially Deprecated and Unmaintained

Location

`NeonEVM/evm_loader/Cargo.lock`

`https://rustsec.org/advisories/RUSTSEC-2020-0036.html`

Synopsis

The EVM loader repository uses the `failure` crate. However, according to the Rust Advisory Database, that particular crate is officially end-of-life. It has security issues with a cvss score of 9.8 (critical).

Remediation

We recommend updating to an actively maintained crate. The Rust Advisory Database has a list of suggested alternatives.

Status

Reported.

Suggestions

Suggestion 1: Consider Using socket2 Instead of net2

Location

`NeonEVM/evm_loader/Cargo.lock`

`https://rustsec.org/advisories/RUSTSEC-2020-0016.html`

Synopsis

The EVM loader repository uses the `net2` crate. However, according to the Rust Advisory Database, that crate has been deprecated and suggests a replacement.

Mitigation

We recommend considering the use of the `socket2` crate, as recommended by the Rust Advisory Database.

Status

Reported.

Suggestion 2: Improve and Increase Code Comments

Location

Example (non-exhaustive):

`NeonEVM/evm_loader`/`program/src/hamt.rs`

Synopsis

A minimal amount of functions in the Neon EVM and the proxy implementation repositories have in-line documentation. Portions of the code critical to the overall design and security of the Neon EVM, such as

the Solana implementation of the hash array mapped trie, are insufficiently explained within the code comments.

Mitigation

We recommend providing short descriptions of every function and explaining crucial parts of the code in the code comments, in order to increase the readability and auditability of the code. In particular, this should be prioritized for low-level sections like the hash array mapped trie implementation.

Status

Reported.

Suggestion 3: Remove Unused Code

Location

Example (non-exhaustive):

NeonEVM/evm_loader/program/src/account_storage.rs#L151

NeonRustEVM/core/src/eval/macros.rs#L2

Synopsis

There are many instances of unused, commented-out code throughout the Neon EVM and proxy implementation repositories, which can cause confusion and misunderstanding for both security researchers and maintainers of the code. This may result in unintended errors that can lead to security vulnerabilities.

Mitigation

We recommend removing outdated code from the codebase and keeping code that is currently unused but might be used in future versions of the project in a separate feature branch.

Status

Reported.

Suggestion 4: Improve Variable and Function Naming

Location

NeonEVM/evm_loader/program/src/solidity_account.rs

NeonEVM/evm_loader/program/src/entrypoint.rs#L234

NeonEVM/evm_loader/program/src/entrypoint.rs#L258

NeonRustEVM/runtime/src/handler.rs#L77

Synopsis

Some of the variable names in the Neon EVM repositories are ambiguous, inconsistent or contain typos. Variables should not be given misleading names. For example, calling Neon EVM native accounts “Solidity accounts” can be misleading as this is just one of several programming languages available for EVM smart contract development. In this case, additional confusion arises when Solidity is abbreviated as `sol`, as this is also the established abbreviation of Solana.

Mitigation

We recommend that variables have correct and unambiguous names. Solidity accounts should be renamed more accurately, for example “Ethereum-native accounts”, and sol should not be used as an abbreviation. Additionally, we recommend that a single and consistent abbreviation for transaction be used.

Status

Reported.

Suggestion 5: Improve Documentation

Location

Neon Labs Documentation: https://doc.neonlabs.org/

Neon EVM `README.md`: https://github.com/LeastAuthority/NeonRustEVM/blob/master/README.md

Synopsis

A high-level technical description and reasoning that explains Rust EVM’s deviation from the original SputnikVM fork is missing. This necessitated avoidable but significant effort in trying to reason about the system. Solana is very different from the traditional blockchain paradigm and the distinguishing characteristics, reasoning, and logic behind the design of the system should be explained in comprehensive documentation.

Robust documentation facilitates a better understanding of the intended system design and behavior by security researchers and maintainers. This reduces the likelihood of confusion or incorrect assumptions and helps in identifying potential inconsistencies or errors in the implementation. In addition, this allows users, maintainers, and security researchers in onboarding to the system more efficiently.

Mitigation

We recommend that the documentation be improved to better describe the logic and rationale for the deviations for the SputnikVM fork, the design objectives of these deviations, and how these objectives are achieved in the implementation.

Status

Reported.

Suggestion 6: Resolve TODOs

Location

Non-exhaustive examples:

`NeonEVM/evm_loader/program/src/account_storage.rs`

`NeonEVM/evm_loader/program/src/executor_state.rs`

`NeonEVM/evm_loader/program/src/executor.rs`

Synopsis

The EVM loader codebase contains many TODOs, few of which suggest notable changes. Unresolved TODO’s raise questions about a system being feature complete. Significant unresolved TODO’s imply

further development to the system, which could alter the security features of the system as reviewed. In addition, unresolved TODOs create confusion about the completeness of the codebase and the intended functionality of each of the system components, which hinders the ability of security researchers to identify implementation errors and security vulnerabilities.

Mitigation

We recommend identifying and resolving any pending TODOs in the codebase.

Status

Reported.

Suggestion 7: Update Solidity Compiler Version

Location

[NeonEVM/evm_loader/SPL_ERC20_Wrapper.sol#L3]

Synopsis

The Solidity compiler version is set as >=0.5.12. Compiling with different versions of the compiler might lead to unexpected results. In addition, older versions of the Solidity compiler contain bugs that have been fixed in more recent versions of the compiler that include up-to-date security patches.

Mitigation

In order to maintain consistency and to prevent unexpected behavior, we recommended that the Solidity compiler version be pinned by removing ">=" , preferably to the latest version of the Solidity compiler.

Status

Reported.

Suggestion 8: Use Explanatory Comments Instead of Declaring Unused Interface

Location

[NeonEVM/evm_loader/SPL_ERC20_Wrapper.sol#L6-L21]

Synopsis

In the Solidity smart contract, `SPL_ERC20_Wrapper`, our team noted that the interface `IERC20` is not used in the implementation. This causes confusion about the state of completeness of the codebase as well as the intended functionality of the system.

Mitigation

We recommend using explanatory code comments to show Application Binary Interface for `delegate call` instead of defining an unused interface to describe functions and events.

Status

Reported.

Suggestion 9: Increase Test Coverage

Location

NeonEVM/evm_loader/program

Synopsis

The smart contracts provided for testing Neon EVM added value in terms of confirming the behaviour of the EVM. However, the EVM loader repository has insufficient unit tests coverage. Unit tests assert the behaviour of the different components, aiding in the understanding of the intended functionality of these components and the system overall. At present, there are only two small test cases implemented for the EVM loader.

Mitigation

We recommend increasing unit test coverage in the EVM loader repository.

Status

Reported.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.