Proximity Labs

Audit

Presented by:

OtterSec contact@osec.io

Max Garrett pew@osec.io

Robert Chen notdeghost@osec.io

Edited by Joshua Charat-Collins

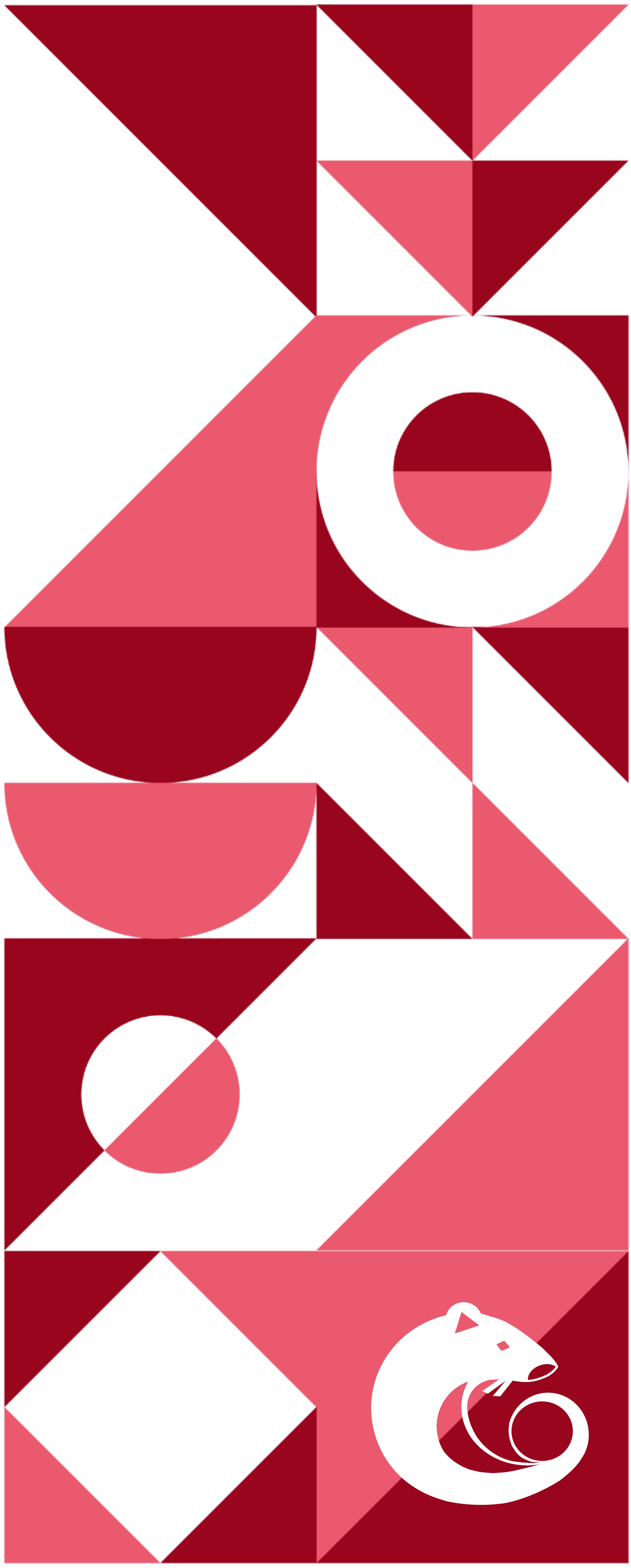


Table of Contents

01 | Executive Summary

Overview

Proximity Labs engaged OtterSec to perform an assessment of the `near-eth` program prior to deployment.

This assessment was conducted between April 27th and May 6th, with a focus on the following:

- Ensure integrity and optimization of the Near-Eth program at an implementation and design level
- Analyze the program attack surface and provide meaningful recommendations to mitigate future vulnerabilities

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the Proximity Labs team to streamline patches and confirm remediation.

Key Findings

The following is a summary of the major findings in this audit.

- 3 findings total
- 1 vulnerability which could lead to loss of funds
    - OS-PRX-ADV-00: Non-context-aware JSON Parsing

As part of this audit, we also provided proof of concepts for each vulnerability to prove exploitability and enable simple regression testing. These scripts can be found at https://osec.io/pocs/near-eth. For a full list, see Appendix C.

02 | Scope

We received the program from Proximity Labs and began the audit April 27th. The source code was delivered to us in a git repository at https://github.com/NearDeFi/near-eth/. This audit was performed against commit 619abeb.

There were a total of three programs included in this audit. A brief description of the two programs is as follows. A full list of program files and hashes can be found in Appendix A.

Name	**Description**
Near-Eth Contract	A minimal no-STD contract that can be deployed to a NEAR account, allowing an Ethereum keypair to control the NEAR account via signing payloads.
Example React App	Example application of a Near-Eth contract client using a React application.

03 | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an onchain program. In other words, there is no way to steal tokens or deny service. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of NEAR's execution model. Some common implementation vulnerabilities include arithmetic overflows and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix B.

One such issue that was identified here was an implementation of custom JSON parsing that would allow users to malform JSON bodies and send unauthorized transactions to NEAR contracts (OS-PRX-ADV-00).

Implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach any target in a team of two. This allows us to share thoughts and collaborate, picking up on details that the other missed.

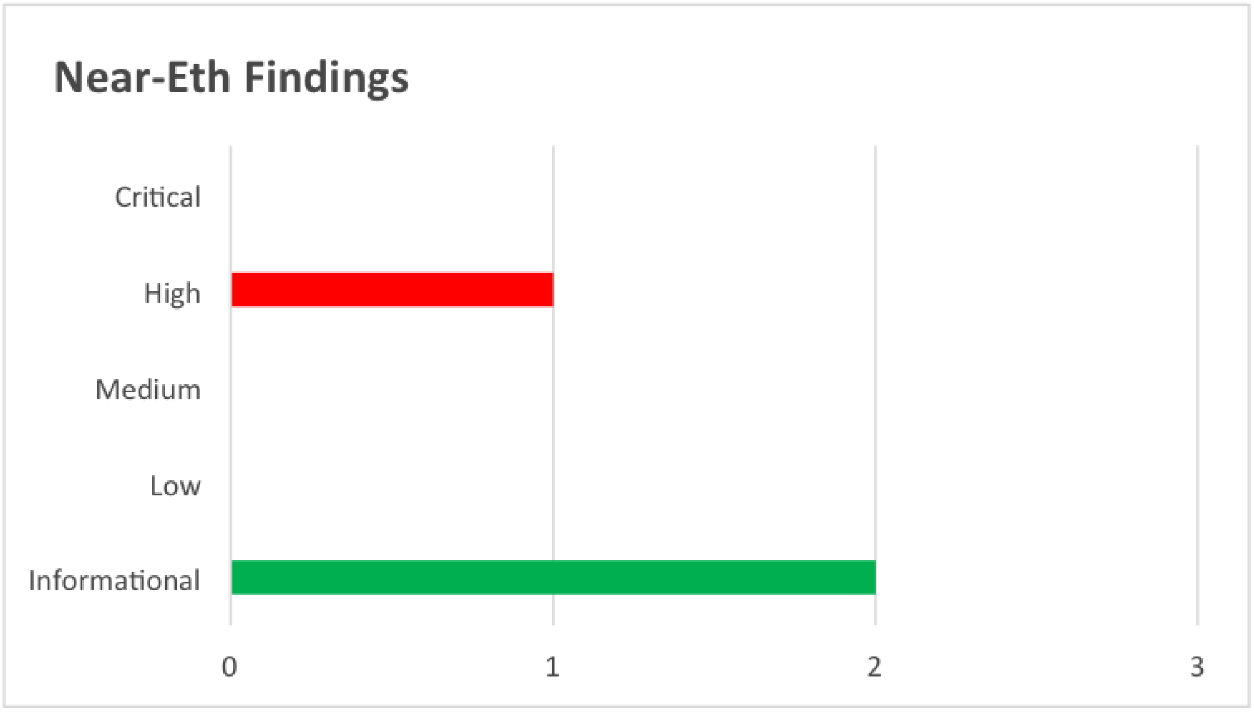
While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

04 | Findings

Overall, we report 3 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



Proof of Concepts

For each vulnerability we created a proof of concept to enable easy regression testing. We recommend integrating these as part of a comprehensive test suite. The proof of concept directory structure can be found in Appendix C.

These proof of concepts can be found at https://osec.io/pocs/near-eth.

To run a POC:

```
./run.sh <directory name>
```

For example,

```
./run.sh os-prx-adv-00
```

See the `README.md` for more information.

05 | Vulnerabilities

Here we present a technical analysis of the vulnerabilities we identified during our audit of the Near-Eth program. These vulnerabilities have **immediate** security implications, and we recommend remediation as soon as possible.

Rating criterion can be found in Appendix E.

ID	Severity	Status	Description
OS-PRX-ADV-00	**High**	**Resolved**	Lack of context-aware JSON parsing allows maliciously formed JSON to affect transactions.

OS-PRX-ADV-00 [High] [Resolved]: Non-context-aware JSON Parsing

Description

Custom JSON parsing is used within the Near-Eth Contract. This parsing method is non-context-aware, which allows for the injection of unintended actions and transactions when transmitting JSON bodies.

As an example, consider an application `X` with a Near account. The Near-Eth Contract is deployed on `X`, and it is providing a service `Y` as a REST API. This API is configured to send `FunctionCalls` to a Near contract, possibly taking user input according to the sample pseudocode provided below:

```
route('/api/call'):
action = get_json_input()
/* JSON Input:
{
                      type: 'FunctionCall',
                      method_name: method_name,
                      args: args,
                      amount: '2.2',
                      gas: '50000000000000',
}
*/
check(action.gas,action.amount)
transaction = {
      nonce: get_c_nonce(),
      transactions: [
          {
              receiver_id: 'arbitraryContract.near',
              actions: [
                  action
              ]
          }
      ]
  }
res = contract.function_call({
  name:'execute'
```

```
    args: gen_args(transaction)
  )
  return res.is_ok()
```

Provided with maliciously-formed JSON input, the onchain program could mistakenly send untrusted actions and transactions. For example, this could allow an attacker to spoof the Transfer action, leading to a loss of user funds. Maliciously-formed JSON input causing a loss of funds is exemplified below:

```
{
      type: 'FunctionCall',
      method_name: 'I_am_a_Function',
      args: '6161',
      amount: '2.2',
      gas: '50000000000000',
      attack_controlled_object: {
            split: 'start]},{end',
            receiver_id: 'attacker.near',
            actions: [{
                        type: 'Transfer',
                        amount: '1337'
                  }]
      }
}
```

This occurs because the user is able to create new key arrays in the transaction and action fields, allowing them to take advantage of the non-context-aware JSON parsing and create unintended actions. Another example of maliciously-formed JSON user input, as well as its result, is shown below:

```
 {
        nonce: 'pew',
        transactions: [
            {
                receiver_id: 'testnet',
                actions: [
                    {
                        type: 'FunctionCall',
                        method_name: 'create_account',
                        args: {
                            "pew": "},{xxd",
                            "type": "GG",
                            "ARGS": "ARE OP"
                        },
                        amount: '2.2',
                        gas: '50000000000000',
                    }
                ]
            }
        ]
    }
```

```
TX: [

"[{\"receiver_id\":\"testnet\",\"actions\":[{\"type\":\"FunctionCall\
",\"method_name\":\"create_account\",\"args\":{\"pew\":\"},{xxd\",\"t
ype\":\"GG\",\"ARGS\":\"ARE
OP\"},\"amount\":\"2.2\",\"gas\":\"50000000000000\"}]}]\"}"

]
Actions: [

"[{\"type\":\"FunctionCall\",\"method_name\":\"create_account\",\"arg
s\":{\"pew\":\"",
"xxd\",\"type\":\"GG\",\"ARGS\":\"ARE
OP\"},\"amount\":\"2.2\",\"gas\":\"50000000000000\"}]}]\"}"

]
```

While this vulnerability does require that a user is able to submit JSON bodies as part of their input, it nevertheless is rated as High due to its potential to spoof highly important actions.

Proof of Concept

1. Attacker accesses service `Y` running on application `X` and uploads malformed JSON data as user input.
2. Attacker utilizes non-context-aware JSON parsing to inject an unintended/malicious transaction action, which is then sent to the Near-Eth contract and processed, leading to loss of funds.

Remediation

Instead of using a custom JSON parsing solution, we recommend storing all values and keys as hex-encoded binaries. Chunks of data would be split by delimiters and decoded using `hex::decode` and `readChunk` in combination. Three context-specific delimiters would be used to separate transactions, actions, and chunks.

Patch

We worked closely with Proximity Labs to rewrite the format of the structured data.

The parsing code now uses length indicators to properly delineate data segments, instead of splitting on JSON delimiters. This makes it much harder for a malicious user to inject their own packets into the data stream.

lib.rs

```
    while transaction_data_copy.len() > 0 {
         let transaction_len: usize =
expect(transaction_data_copy[HEADER_OFFSET..HEADER_SIZE].parse().ok()
);
         transaction_data_copy =
&transaction_data_copy[PAYLOAD_START+transaction_len..];
         num_txs += 1;
    }
```

We also worked with the team to introduce a receivers list at the start of the payload to make it much harder for an adversary to hide malicious receivers. This way, a user can easily inspect the payload and identify which addresses are involved in the transaction.

lib.rs

```
      receivers_data =
&receivers_data[PAYLOAD_START..PAYLOAD_START+receivers_len];
      let mut receivers: Vec<&str> =
receivers_data.split(",").collect();
      if num_receivers != receivers.len() || receivers_len !=
receivers_data.len() {
            sys::panic();
      }
```

06 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they do represent antipatterns and could introduce a vulnerability in the future.

ID	Description
OS-PRX-SUG-00	`contract/src/lib.rs` contains inefficient string splitting procedures.
OS-PRX-SUG-01	Rust compiler flags could be changed for optimization's sake.

OS-PRX-SUG-00: Optimizing Using Iteration

Description

There is a more efficient way to separate the strings in `transactions_split` and `actions_split` than by calling `split()` and `collect()` on them sequentially. As currently programmed, this increases the gas use of the program and could be optimized.

Remediation

On lines 138 and 147 of `contract/src/lib.rs`, the following changes could be made. On line 138, the following code:

contract/src/lib.rs

```
let transactions: Vec<&str> =
transactions_split.split("]},{").map(|x| x.trim()).collect();
```

should be changed to:

contract/src/lib.rs (revised)

```
let transactions: Vec<&str> =
transactions_split.into_iter("]},{").map(|x| x.trim());
```

On line 147:

contract/src/lib.rs

```
let actions: Vec<&str> = actions_split.split("},{").map(|x|
x.trim()).collect();
```

should be changed to:

contract/src/lib.rs (revised)

```
let actions: Vec<&str> = actions_split.in_iter("},{").map(|x|
x.trim());
```

This should reduce gas usage in the program.

OS-PRX-SUG-01: Optimized Rust Compiler Flags

Description

Some of the compiler flags used for Rust could be changed in order to optimize the program further.

Remediation

Change the following compiler flags to the Rust compiler configuration file from the following:

contract/Cargo.toml

```
[profile.release]
codegen-units = 1
opt-level = "s"
lto = true
debug = false
panic = "abort"
overflow-checks = true
```

to this. Note that `overflow-checks` has been removed, `strip` has been added, and optimization level has been set to "`z`".

contract/Cargo.toml (revised)

```
[profile.release]
codegen-units = 1
opt-level = "z"
lto = true
debug = false
strip = true
panic = "abort"
```

This results in the program being 1191 bytes smaller.

07 | Appendix

Appendix A: Program Files

Below are the files in scope for this audit and their corresponding sha256 hashes.

```
env                                       998ee2a581916223d18ff3dde1a3193e
.eslintrc.js                               fc553e787828ddcefc2adba0f5ff940d
README.md                                  41f98919427540146acea826ba16a2be
package.json                               d29cdb9fa1ad22b91854204d0d018493
yarn.lock                                  a997a9a4085bf191302ddc6b04c3546c
account-map
  Cargo.lock                               387453acdd086e204a7b49ced5c52b7d
  Cargo.toml                               eff69faf140afcd9505b1148b005b008
  build.sh                                 81a9c92129e63e676de3091aacd5930c
  src
    lib.rs                                 e489c6f4cc2eab7372d8c88b32e88444
contract
  Cargo.lock                               29542b7d484b710c9b03baf35b57c9cd
  Cargo.toml                               cc701d6895cc9bd636eefeefd312c730
  build.sh                                 995dcb469ca1441ee599d4d0dc7939a1
  rust-toolchain.toml                      696b0c249ad153d627d4426a0d519fe9
  .cargo
    config.toml                            1911d71a7e234e232962a8374334d5cf
  src
    lib.rs                                 520aa105a8e853b81802b3e04663d811
    owner.rs                               e9894b547c91ccf79492423d45b964ba
    parse.rs                               843a172ea2a5139d2d7f329ccb789656
    sys.rs                                 d15f2032ff0fab4d5b676fd98057bcdb
src
  App.js                                   883e624debceb157e6294fb6917dfa4d
  App.scss                                 c0910aaea20e0128c4cab1918b3dee08
  HelloMessage.js                          9b2333bee8c76462284b890444ae524c
  index.html                               df5db6fb74bd712d5a283ac9212bca77
  index.js                                 5e26e7100daa46d9cc14a5fd1d7bcbb7
  state
    app.js                                 8ed7347ee159af29467656997ce4a16e
  utils
    neth.js                                d28bfe1173f25d286c65817a88a5cef4
    state.js                               b5c620ad0b1bda0839d7dda047f6dcc6
    store.js                               4189bb683aa3f7c069964c85a1b17827
test
  contract.test.js                         fcf22985d127c82fb24371587fcca843
  test-utils.js                            a8b31bdef4936aaf73f8cd5b6da9f892
utils
  config.js                                af6a064fe281d41cacc86e53cb357a4f
```

near-utils.js	6f048a07b362f1169768e46d095e82b7
patch-config.js	00c4a3a4df28bb503b7284d177110f73

Appendix B: Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust `as` conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Input Validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so.
Numbers	Reasonable limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have reasonable size restrictions to prevent denial of service conditions.

Miscellaneous

Libraries	Out-of-date libraries should not include any publicly disclosed vulnerabilities.
Clippy	`cargo clippy` is an effective linter to detect potential anti-practices.

Appendix C: Proof of Concepts

Below are the provided proof of concept files and their sha256 hashes.

```
README.md                               82d17d5e5c88a71151f55ab4bff147c8
build.sh                                b9d0dc2d8595bc278c80886ef5196234
package.json                            e083a0d2d1c5f9394c35f29230996b3f
reset-localnet.sh                       6fc21ef9ba933372bba8b6270db333fa
run.sh                                  ccbbee51cc097c9d16a5ce3ade6203de
framework
  framework.js                          519877a065335cfa017837ac8f0aa755
  near-eth-utils.js                     98261b2f364c6a3f3407d56379dd8ab0
os-prx-adv-00
  poc.js                                48ffb1ce7e6dde754ecf0d8ffec8c396
```

Appendix D: Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities which **immediately** lead to loss of user funds with minimal preconditions Examples: - Misconfigured authority/token account validation - Rounding errors on token transfers
High	Vulnerabilities which **could** lead to loss of user funds but are potentially difficult to exploit. Examples: - Loss of funds requiring specific victim interactions - Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities which could lead to denial of service scenarios or degraded usability. Examples: - Malicious input cause computation limit exhaustion - Forced exceptions preventing normal use
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: - Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as *general findings*. Examples: - Explicit assertion of critical internal invariants - Improved input validation - Uncaught Rust errors (vector out of bounds indexing)