Guvenkaya® The Bedrock of Security

Trove Labs - Neko

Airdrop Script Security & Code Review

Lead Security Engineer: Timur Guvenkaya
Lead Software Engineer: Aimen Sahnoun
Date of Engagement: 3rd April 2024 - 8th April 2024
Visit: www.guvenkaya.co

Contents

About Us

Guvenkaya is a security research firm specializing in Rust security, Web3 security of Rust-based protocols, and Web2 security. With our expertise, we provide both security auditing services and custom security solutions

About Neko

NEKO is NEAR Protocol's first meme coin, a crypto content creation hub and AI app developer. NEKO was born in early 2022 as the first meme coin on NEAR Protocol, aiming to become a key community support node. The Neko community was cultivated during the most challenging market conditions that NEAR has ever seen. When the chips were down, the Neko community grew stronger; showing up and creating educational crypto content every day. By the end of 2023, Neko was making waves. Growing into one of the largest NEAR native communities with over 23,500 unique holders!

Audit Results

Guvenkaya conducted a security and code improvement assessment of the NEKO airdrop script from 3rd April 2024 to 8th April 2024. During this engagement, a total of 11 findings were reported. 3 of the findings were medium, and the remaining were informational severity. All major issues are fixed by the Trove Labs team.

Project Scope

Lines of Code Reviewed: 204

Airdrop Script

File name	Link
Index	https://github.com/Trove-team/airdrop-tool/blob/main/airdrop-tool-mac/src/index.ts

Out of Scope

The audit will include, but is not limited to, reviewing the code for security vulnerabilities, coding practices, and architecture. The audit does not include a review of the dependencies.

Repo with Fixes

The Trove Labs Team has rewritten the airdrop script and fixed the issues in the following repo: https://github.com/Good-Fortune-Felines-Core-Team/airdrop-tool

Timeline

Start of the audit	Draft report	Final report
3rd April 2024	10th April 2024	8th June 2024

Methodology

- RESEARCH INTO PROJECT ARCHITECTURE
- PREPARING ATTACK VECTORS
- SETTING UP AN ENVIRONMENT
- MANUAL CODE REVIEW OF THE CODE
- ASSESSMENT OF RUST SECURITY ISSUES
- ASSESSMENT OF NEAR SECURITY ISSUES
- ASSESSMENT OF ARITHMETIC ISSUES
- BUSINESS LOGIC VULNERABILITY ASSESSMENT
- ONCHAIN TESTING USING NEAR WORKSPACES
- BEST PRACTICES AND CODE QUALITY
- CHECKING FOR CODE REFACTORING/SIMPLIFICATION POSSIBILITIES
- ARCHITECTURE IMPROVEMENT SUGGESTIONS
- PREPARING POCS AND/OR TESTS FOR EACH CRITICAL/HIGH/MEDIUM ISSUES

Severity Breakdown

01. Likelihood Ratings

Likely: The vulnerability is easily discoverable and not overly complex to exploit.
Possible: The vulnerability presents some challenges either in discovery or in the complexity of the attack.
Rare: The vulnerability is either very difcult to discover or complex to exploit, or both.
This matrix provides a nuanced view, taking into account both the ease of discovering a vulnerability and the complexity involved in exploiting it.

02. Impact

Severe: The vulnerability is easily discoverable and not overly complex to exploit.
Moderate: The vulnerability presents some challenges either in discovery or in the complexity of the attack.
Negligible: The vulnerability is either very difcult to discover or complex to exploit, or both.

03. Severity Ratings

Critical: Assigned to vulnerabilities with severe impact and a likely likelihood of exploitation.
High: For vulnerabilities with either severe impact but only a possible likelihood, or moderate impact with a likely likelihood.
Medium: Used for vulnerabilities with severe impact but a rare likelihood, moderate impact with a possible likelihood, or negligible impact with a likely likelihood.
Low: For vulnerabilities with moderate impact and rare likelihood, or negligible impact with a possible likelihood.
Informational: The lowest severity rating, typically for vulnerabilities with negligible impact and a rare likelihood of exploitation.

CRITICAL HIGH MEDIUM Low Informational

Likelihood Matrix:

Attack Complexity \ Discovery Ease	Obvious	Concealed	Hidden
Complex	Possible	Rare	Rare
Moderate	Likely	Possible	Rare
Straightforward	Likely	Possible	Possible

Likelihood/Impact Matrix:

Likelihood \ Impact	Severe	Moderate	Negligible
Likely	**CRITICAL**	**HIGH**	**MEDIUM**
Possible	**HIGH**	**MEDIUM**	Low
Rare	**MEDIUM**	Low	Informational

Findings Summary

01. Remediation Complexity: This measures how difcult it is to fx the vulnerability once it has been identifed.

Simple: Patches or fixes are readily available and easily implemented.
Moderate: Requires some time and resources to remediate, but well within the capabilities of most organizations.
Difficult: Remediation requires significant resources, specialized skills, or substantial changes to systems or architecture.

02. Status: This measures how difcult it is to fx the vulnerability once it has been identifed.

Not Fixed: Indicates that the vulnerability has been identifed but no remedial action has been taken yet. This status is crucial for newly discovered vulnerabilities or those awaiting prioritization.
Fixed: This status is applied when the vulnerability has been successfully remediated. It implies that appropriate measures (like patching, confguration changes, or architectural modifcations) have been implemented to resolve the issue.
Acknowledged: This status is used for vulnerabilities that have been recognized, but for various reasons (such as risk acceptance, cost, or other business decisions), have not been fxed. It indicates that the risk posed by the vulnerability is known and has been consciously accepted.

Finding	Impact	Likelihood	Severity	Remediation Complexity	Remediation Status
GUV-1: Lack of Error Handling in Airdrop Script	Moderate	Possible	**MEDIUM**	Simple	Fixed
GUV-2: Several Issues in The Promise Handling	Moderate	Possible	**MEDIUM**	Simple	Fixed
GUV-3: Finished Folder Is Always Empty	Moderate	Possible	**MEDIUM**	Simple	Fixed
GUV-4: Repetition and Lack of Modularity in the Script	Negligible	Rare	Informational	Moderate	Fixed
GUV-5: Insufficient Retry Functionality In The Script	Negligible	Rare	Informational	Moderate	Fixed
GUV-6: Redundant Script Folder Should Be Removed	Negligible	Rare	Informational	Simple	Fixed
GUV-7: Redundant RPC Call To Check Whether The Account Exists	Negligible	Rare	Informational	Moderate	Fixed
GUV-8: Lack Of Feedback On Completed Transactions	Negligible	Rare	Informational	Simple	Fixed
GUV-9: Missing Zero Value Check On Airdrop Amount	Negligible	Rare	Informational	Simple	Acknowledged
GUV-10: Error File Should Be Replaced Instead Of Appended Into	Negligible	Rare	Informational	Simple	Fixed
GUV-11: Insufficient Documentation	Negligible	Rare	Informational	Simple	Fixed

Findings Details

GUV-1 Lack of Error Handling in Airdrop Script - Medium

We noticed that the airdrop script does not handle errors originating from the smart contract. For instance, if we try to send more than the available balance, the script won't fail. We will get a false sense of success. In addition, the script used both try/catch and .then/.catch syntaxes, leading to inconsistent code and difficulty in reading the code. There was also the introduction of more detailed and specific errors for all async operations. For example, if there is an error on reading a JSON file, we get the following error: **Error reading JSON file: $error** , whereas before, there was general errors such as **Retrying trasnfer to : $id**

PROPOSED SOLUTION

We propose logging smart contract fails for each account and moving those account to error folder.

REMEDIATION - FIXED

The Trove Labs team has fixed the issue in the following releases: **v1.1.0** & **v1.2.0**

GUV-2 Several Issues In The Promise Handling - Medium

We noticed that in promises, there was the use of the resolve method for all cases, even when there is an error, the syntactically correct method to use is rejected when there is an error within a promise. There was also a misleading log that says **Retrying transfer to id : $id** , where then the Promise is resolved and the code moves on to the next Promise, in this sense, there is no retry logic implemented. The use of Promise.all while improving performance; it can introduce a wide variety of issues due to its parallel execution nature.

NONCE MANAGEMENT

Since the script is manually managing the nonce, using Promise.all can cause failures because of its non sequential nature, Which can lead to nonce conflicting and eventually cause failure.

ERROR HANDLING

Due to its parallel execution nature, error handling can be difficult to manage.

script:runMain:airdrop-tool-mac/src/index.ts

```
          await fs.writeFile(
            join(__dirname, "..", "data", listName + ".json"),
            JSON.stringify(list),
          );

        resolve();
        } catch (e) {
        console.log("Retrying Transfer to ", id);
        resolve();
        }
      });
      index++;
      promiseArray.push(promise);
    }
  await Promise.all(promiseArray);
```

PROPOSED SOLUTION

We propose refactoring the code to ensure that the Promise errors are properly handled and that the nonce is managed correctly.

REMEDIATION - FIXED

The Trove Labs team has fixed the issue in the following release: **v1.0.0**

GUV-3 Finished Folder Is Always Empty - Medium

We noticed that after the completion of the airdrop, the json file which is supposed to contain the list of accounts that have been airdropped is empty. It happens because after the verification of the account existence, the account is deleted from the original file. Since the original file is used to write the list of accounts that have been airdropped to the **final** folder, the file is empty.

script:runMain:airdrop-tool-mac/src/index.ts

```
          const accountData = await nearConnection.connection.provider
           .query(`account/${id}`, "")
           .catch(async (_accountError) => {
             console.log("Invalid account Id:", id);
             await fs.appendFile(
               join(__dirname, "..", "error", listName + ".txt"),
               `${id}:${transferAmount.toString()}

    `,
             );
           delete list[id];
           await fs.writeFile(
             join(__dirname, "..", "data", listName + ".json"),
             JSON.stringify(list),
             );
           });
```

PROPOSED SOLUTION

We propose either refactoring the application to avoid deleting the date from the original list or directly on each iteration appending to the file in the **finished** folder after clearing the old data.

REMEDIATION - FIXED

The Trove Labs team has fixed the issue in the following release: **v1.0.0**

GUV-4 Repetition and Lack of Modularity in the Script - Informational

We noticed that the original file had a lot of repetitive blocks in terms of prompting the user and file handling. This makes it harder to read, debug, and implement changes to the script, increasing the risk of introducing bugs by mistake.

PROPOSED SOLUTION

Consider refactoring the code to make it more modular and reduce repetition. Example of refactored script is shown below:

script:runMain:airdrop-tool-mac/src/index.ts

```
import { connect, Contract, keyStores, utils, transactions } from "near-apijs";
import inquirer from "inquirer";
import fs from "fs/promises";
import os from "os";
import path from "path";
import { fileURLToPath } from "url";
import BN from "bn.js";
// Custom types =====================
type TokenMethod = {
  ft_balance_of: (args: { account_id: string }) => Promise<string>;
  storage_balance_of: (args: { account_id: string }) => Promise<any>;
  ft_transfer: (args: {
    args: { receiver_id: string; amount: string };
    amount: string;
  }) => Promise<void>;
  storage_deposit: (args: {
    args: { account_id: string; registration_only: boolean };
    amount: string;
  }) => Promise<string>;
};
```

script:runMain:airdrop-tool-mac/src/index.ts

```
type TokenContract = Contract & TokenMethod;
type List = {
  [walletId: string]: number;
};
// ==============================
// Global variables ===========================
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
const homedir = os.homedir();
const CREDENTIALS_DIR = ".near-credentials";
const credentialsPath = path.join(homedir, CREDENTIALS_DIR);
const myKeyStore = new
keyStores.UnencryptedFileSystemKeyStore(credentialsPath);
const finishedList: List = {};
// ============================
// Configs ===============
const CONFIG = {
  networkId: "mainnet",
  nodeUrl: "https://rpc.mainnet.near.org",
  walletUrl: "https://wallet.mainnet.near.org",
  helperUrl: "https://helper.mainnet.near.org",
  explorerUrl: "https://explorer.mainnet.near.org",
  keyStore: myKeyStore,
};
const contractId = "ndc.tkn.near";
const PROMPT_MESSAGES = {
  accountId: "Enter Account Id used for shooting Token:",
  listName: "Enter the json file name:",
  amount: "Enter the amount of TOKEN per NFT to shoot:",
  confirm: (listName: string) =>
    `Start Shooting Token to the list [ ${listName} ]?`,
};
```

script:runMain:airdrop-tool-mac/src/index.ts

```
// ==========
// Helper methods =================
async function promptUser(
  message: string,
  type: string = "input",
  name: string = "input",
): Promise<string> {
  const { [name]: value } = await inquirer.prompt([{ type, name, message }]);
  return value;
}
async function getContract(account: any): Promise<TokenContract> {
  return new Contract(account, contractId, {
    viewMethods: ["ft_balance_of", "storage_balance_of"],
    changeMethods: ["ft_transfer", "storage_deposit"],
  }) as TokenContract;
}
async function readJsonFile(
  filePath: string,
): Promise<{ [walletId: string]: number }> {
  try {
    const data = await fs.readFile(filePath, "utf-8");
    return JSON.parse(data);
  } catch (error) {
    console.error("Error reading JSON file:", error);
    throw error;
  }
}
```

script:runMain:airdrop-tool-mac/src/index.ts

```
async function writeJsonFile(filePath: string, data: object): Promise<void> {
  try {
    await fs.writeFile(filePath, JSON.stringify(data, null, 2));
  } catch (error) {
    console.error("Error writing JSON file:", error);
    throw error;
  }
}
async function appendToFile(filePath: string, content: string): Promise<void> {
  try {
    await fs.appendFile(filePath, content);
  } catch (error) {
    console.error("Error appending to file:", error);
    throw error;
  }
}
async function moveFile(source: string, destination: string): Promise<void> {
  try {
    await fs.rename(source, destination);
  } catch (error) {
    console.error("Error moving file:", error);
    throw error;
  }
}
// =================
```

script:runMain:airdrop-tool-mac/src/index.ts

```
async function processTransfers(
  account: any,
  contract: TokenContract,
  listName: string,
  amount: BN,
): Promise<void> {
  const listFilePath = path.join(__dirname, "..", "data", `${listName}.json`);
  const list: List = await readJsonFile(listFilePath);
  const signerPubKey = await account.connection.signer.getPublicKey(
    account.accountId,
    CONFIG.networkId,
  );
  const accessKey: any = await account.connection.provider.query(
    `access_key/${account.accountId}/${signerPubKey}`,
    "",
  );
  let nonce = accessKey.nonce + 1;
  for (const [walletId, balance] of Object.entries(list)) {
    try {
      const transferAmount = new BN(balance).mul(amount);
      // Check if the account exists
      try {
        await account.connection.provider.query(`account/${walletId}`, "");
      } catch (error) {
        console.log("Invalid account Id:", walletId);
        await appendToFile(
          path.join(__dirname, "..", "error", `${listName}.txt`),
          `${walletId}:${transferAmount.toString()}
`,
        );ß
        delete list[walletId];
        await writeJsonFile(listFilePath, list);
        continue;}
```

script:runMain:airdrop-tool-mac/src/index.ts

```
console.log("Valid account Id:", walletId);
console.log("Checking Storage Balance...");
const storageBalance = await contract.storage_balance_of({
  account_id: walletId,
});
const actions = [];
if (!storageBalance) {
  console.log("Registering Storage", walletId);
  actions.push(
    transactions.functionCall(
      "storage_deposit",
      { account_id: walletId, registration_only: true },
      new BN("30000000000000"),
      new BN(utils.format.parseNearAmount("0.00125")!),
    ),
  );
}
actions.push(
  transactions.functionCall(
    "ft_transfer",
    { receiver_id: walletId, amount: transferAmount.toString() },
    new BN("30000000000000"),
    new BN("1"),
  ),
);
const recentBlockHash = utils.serialize.base_decode(accessKey.block_hash);
const transaction = transactions.createTransaction(
  account.accountId,
  signerPubKey,
  contractId,
  nonce++,
  actions,
  recentBlockHash,
);
```

script:runMain:airdrop-tool-mac/src/index.ts

```
          console.log("Transferring to", walletId);
          const signedTransaction = await transactions.signTransaction(
            transaction,
            account.connection.signer,
            account.accountId,
            CONFIG.networkId,
          );
          await account.connection.provider.sendTransaction(signedTransaction[1]);
          finishedList[walletId] = list[walletId];
          delete list[walletId];
          await writeJsonFile(listFilePath, list);
        } catch (error) {
          console.error(
            `Error processing transfer to wallet :
    ${walletId} with error:`,
            error,
          );
        }
      }
      if (Object.keys(list).length === 0) {
        const finishedFilePath = path.join(
          __dirname,
          "..",
          "finished",
          `finished_${listName}.json`,
        );
        await writeJsonFile(finishedFilePath, finishedList);
        console.log("✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔  DONE ✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔ ");
      } else {
        console.log(
          "========================== Partially Done
==============================",
        );
        await processTransfers(account, contract, listName, amount);}}
```

script:runMain:airdrop-tool-mac/src/index.ts

```
async function main() {
  const nearConnection = await connect(CONFIG);
  const accountId = await promptUser(PROMPT_MESSAGES.accountId);
  const account = await nearConnection.account(accountId);
  const contract = await getContract(account);
  const listName = await promptUser(PROMPT_MESSAGES.listName);
  const amount = new BN(await promptUser(PROMPT_MESSAGES.amount));
  const confirmTransfer = await promptUser(
    PROMPT_MESSAGES.confirm(listName),
    "confirm",
  );
  if (!confirmTransfer) {
    console.log("Okay, bye!");
    return;
  }

  await processTransfers(account, contract, listName, amount);
}
main().catch((error) => {
  console.error("Error in main:", error);
  process.exit(1);
});
```

REMEDIATION - FIXED

The Trove Labs team has fixed the issue in the following release: **v1.0.0**

GUV-5 Insufficient Retry Functionality In The Script - Informational

We noticed that the script does not have a proper retry mechanism with backoff strategy in case of a failure. Transactions can fail due to RPC issues such as timeout error and invalid nonce errors. It is crucial to ensure that those transactions are repeated **MAX_REPEAT_TIMES**.

PROPOSED SOLUTION

Make sure to implement a retry mechanism with a backoff strategy to handle transaction failures. It is crucial to only retry RPC related errors since errors originating from the smart contract cannot be resolved by retrying.

REMEDIATION - FIXED

The Trove Labs team has fixed the issue in the following releases: **v1.0.0** & **v1.1.0**

GUV-6 Redundant Script Folder Should Be Removed - Informational

We noticed that both airdrop-tool-mac and airdrop-tool-windows contain exactly the same script. This redundancy can lead to confusion and potential issues in the future. It is recommended to remove one of the folders and keep only one version of the script.

PROPOSED SOLUTION

Consider leaving a single folder and renaming it to airdrop-tool.

REMEDIATION - FIXED

The Trove Labs team has fixed the issue in the following release: **v1.0.0**

GUV-7 Redundant RPC Call To Check Whether The Account Exists - Informational

We noticed that the script calls an RPC to check whether the account is valid and exists. However, it is possible to avoid this call by checking on Transaction error returned. If error relates to the accunt not being found, then the account does not exist. The only thing we can do beforehand is to check whether the account id follows the correct format, which is fast.

script:runMain:airdrop-tool-mac/src/index.ts

```
           const accountData = await nearConnection.connection.provider
           .query(`account/${id}`, "")
            .catch(async (_accountError) => {
              console.log("Invalid account Id:", id);
              await fs.appendFile(
                join(__dirname, "..", "error", listName + ".txt"),
                `${id}:${transferAmount.toString()}

  `,
              );
              delete list[id];
              await fs.writeFile(
                join(__dirname, "..", "data", listName + ".json"),
                JSON.stringify(list),);});
            ...
```

PROPOSED SOLUTION

Consider matching on error after the transaction is sent and beforehand check whether the account id format correct.

REMEDIATION - FIXED

The Trove Labs team has fixed the issue in the following release: **v1.1.0**

GUV-8 Lack Of Feedback On Completed Transactions - Informational

We noticed that after the transaction is sent we do not get any feedback whether it was successful or failed. Also, we do not get a transaction hash which can be used to check the status later or share with the user

PROPOSED SOLUTION

Consider logging whether the transaction was successful or failed with a link to a block explorer where the user can check the status of the transaction if needed.

REMEDIATION - FIXED

The Trove Labs team has fixed the issue in the following release: **v1.0.0**

GUV-9 Missing Zero Value Check On Airdrop Amount - Informational

We noticed that script operator is allowed to put zero as the number of airdrops. Having zero as the number of airdrops leads to the redundant execution of the script. It is recommended to add a check to prevent the operator from executing the script with zero airdrops.

PROPOSED SOLUTION

Consider adding a check to prevent the operator from executing the script with zero as the number of airdrops.

REMEDIATION - ACKNOWLEDGED

The Trove Labs has acknowledged the issue.

GUV-10 Error File Should Be Replaced Instead Of Appended Into - Informational

We noticed that ther error txt file is appended on each run. This can lead to duplicate entries if the script is executed multiple times. It is recommended to replace the file on each run.

PROPOSED SOLUTION

Consider replacing the error file on each run instead of appending to it.

REMEDIATION - FIXED

The Trove Labs team has fixed the issue in the following release: **v1.0.0**

GUV-11 Insufficient Documentation - Informational

We noticed that the documentation does not mention that the script requires both **error** and **final** folders to be present in the root directory. It is recommended to add this information to the documentation to avoid any confusion.

PROPOSED SOLUTION

Consider adding information to the documentation that the script requires both **error** and **final** folders to be present in the root directory.

REMEDIATION - FIXED

The Trove Labs team has fixed the issue in the following release: **v1.0.0**