Neatar Security Analysis

by Pessimistic

This report is public

December 1, 2021

Abstract

In this report, we consider the security of smarts contracts of Neatar project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, security audit is not an investment advice.

Summary

In this report, we considered the security of Neatar project smart contracts, available on a public GitHub repository. We performed our audit according to the procedure described below.

The audit showed four issues of high severity. Also, several low-severity issues were found.

The only provided documentation for the project is a readme.md file in the repository. The code base is thoroughly covered with tests.

Project overview

Project description

For the audit, we were provided with Neatar project smart contracts on a public GitHub repository, commit 6747cfbd64238585ecefbe2595ae040a0dcacf92.

The repository contains the **README.md** file.

The `Neatar` contract complies with `Core` and `Metadata` sections of NEP-171 NFT standard.

The code base is thoroughly covered with tests.

Review procedure

We perform our audit according to the following procedure:

Automated analysis

- We compile the contracts.
- We scan the project's codebase with Clippy static analyzer.
- We manually verify (reject or confirm) all the issues found by the tool.
- We run Valgrind on the contract in emulated environment.

Manual analysis

- We manually review the code and assess its quality.
- We check the code for known vulnerabilities.
- We check whether the code logic complies with the provided documentation.
- We suggest possible gas and storage optimizations.

Issues

We are actively looking for:

- Access control issues (incorrect admin or users identification/authorization).
- Lost/stolen assets issues (assets being stuck on the contract or sent to nowhere or to a wrong account).
- DoS due to logical issues (deadlock, state machine error, etc).
- DoS due to technical issues (Out of Gas error, other limitations).
- Contract interaction issues (reentrancy, insecure calls, caller assumptions).
- Arithmetic issues (overflow, underflow, rounding issues).
- Incorrect Near SDK usage.
- Other issues.

Automated analysis

Automated analysis did not show any issues:

- We run `Clippy` static analysis tool on the project's source code. It did not discover any issues, since the team uses it regularly.
- We run `Valgrind` on the provided unit tests. It did not detect any memory issues.

Manual analysis

The audit showed several issues in **neatar.rs** source file. All these issues are listed below.

High severity issues

High severity issues severely disrupt, cripple, or otherwise violate the contract's behavior. We do not recommend deploying or using the contracts with any issues of this severity.

Bug

```
176  and_then(|per_owner| per_owner.remove(&owner_id))
```

The `ft_burn()` function is supposed to remove a single token by its ID. However, the code removes the whole list of tokens that belong to the user. It leaves the storage of the contract in an inconsistent state.

Bad design

```
209  let owner_id = env::signer_account_id();
```

The contract grants an avatar (token) to the signer of the transaction, rather than to the caller (i.e., `env::predecessor_account_id()`) of `avatar_create()` function. It prohibits complex flows with contract interactions and limits overall usefulness of the project. We recommend always interacting with the caller regardless of the call stack.

Limited avatar creation rate

```
227  &env::sha256(format!("{}-{}", owner_id,
     env::block_timestamp()).as_bytes());
```

Only one avatar can be created by a user within a single block. This restriction seems unreasonable. It prohibits complex flows with contract interactions and limits the overall usefulness of the project.

Non-approved avatar change

The user's avatar can be changed without an approval from the user, since anyone can transfer a Neatar token to any other account, and the last received token is used as the avatar.

Low severity issues

Low severity issues do not directly affect project operation. However, they might lead to various problems in the future versions of the code. We recommend taking them into account.

Excessive gas consumption

```
160  let list = self.token.nft_tokens_for_owner(account_id, None,
     None);
```

The list of user's tokens is uploaded to a local variable. However, only the last value is required. Therefore, the function becomes significantly inefficient for users who possess a large number of Neatar tokens.

Insufficient error handling

```
203  let media = token.metadata.unwrap().media.unwrap();
```

The code will call panic in case of metadata absence. This is not an issue for the current implementation. However, we recommend writing the code that falls gracefully.

Unnecessary assignments

The contract uses a deprecated `NonFungibleToken.mint()` function which includes the `token.owner_id` check. To pass this check, `avatar_create_for()` function assigns `env::predecessor_account_id()` value to the `owner_id` variable every time it is called.

These multiple assignments are unnecessary. The variable can be initialized only once and remain unchanged since it is always the contract's account used in the current implementation. Anyway, we recommend using the latest Non-fungible token version instead.

This analysis was performed by Pessimistic:

Sergey Grigoriev, Security Engineer
Evgeny Marchenko, Senior Security Engineer
Boris Nikashin, Analyst
Alexander Seleznev, Founder

December 1, 2021