HackCtrl

# SMART CONTRACT
# SECURITY ANALYSIS

| | |
|---|---|
| **Report for:** | EDDA |
| **Date:** | 8 March 2021 |

## Document:

| | |
|---|---|
| Name | Smart Contract Code Review and Security Analysis Report for Customer |
| Platform | |
| Link | |
| Date of the first audit | 08.03.2021 |
| Version of the first audit | commit ada4e132f552aec63933d00b6bc9b4fc1d988323 |
| Date of the secondary audit | |
| Version of the secondary audit | |


# *Table of contents*


| | |
|---|---|
| Introduction | 4 |
| Scope | 4 |
| Methodology | 6 |
| Executive Summary | 7 |
| Severity Definitions | 7 |
| AS-IS overview | 8 |
| Audit overview | 11 |
| Conclusion | 14 |
| Disclaimers | 15 |

# Introduction

HackControl (Consultant) was contracted to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of Customer's smart contract and its code review conducted between February 25th, 2021 – March 8th, 2021.

The objectives of the audit are as follows:

1. Determine correct functioning of the contract, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine contract bugs, which might lead to unexpected behaviour.
4. Analyse whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents the summary of the findings. As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

# Scope

The code has been provided by the developers in the form of private GitHub repository with the last commit hash value:

ada4e132f552aec63933d00b6bc9b4fc1d988323

We have scanned this smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that are considered (the full list includes them but is not limited to them):


- Reentrancy
- Ownership Takeover
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Style guide violation
- Costly Loop
- ERC20 API violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level
- Deployment Consistency
- Repository Consistency
- Data Consistency
- Business Logics Review
- Functionality Checks
- Access Control & Authorization
- Escrow manipulation
- Token Supply manipulation
- Assets integrity
- User Balances manipulation
- Kill-Switch Mechanism
- Operation Trails & Event Generation


# Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the contract's intended purpose by reading the available documentation and codebase

2. Automated scanning of the contract with static code analysis tools for security vulnerabilities and use of best practice guidelines.

- we scan project's smart contracts with several publicly available automated Solidity analysis tools such as Remix, Mythril and Solhint
- we manually verify (reject or confirm) all the issues found by tools

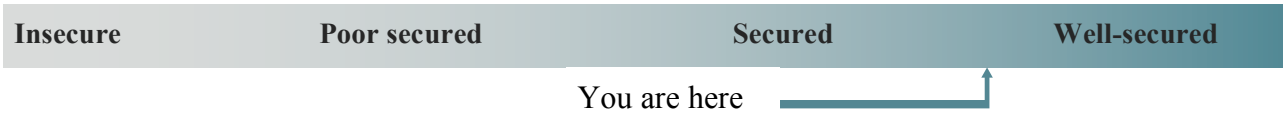
3. Manual line by line analysis of the contracts source code for security vulnerabilities and use of best practice guidelines, including but not limited to:

- Reentrancy analysis
- Race condition analysis
- Front-running issues and transaction order dependencies
- Time dependencies
- Under- / overflow and math precision issues
- Function visibility Issues
- Possible denial of service attacks
- Storage Layout Vulnerabilities

4. Report writing

# Executive Summary

According to the assessment, Customer's smart contracts are well secured.




Our team performed analysis of code functionality, manual audit and automated checks with solc, Mythril, Slither and remix IDE. All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the “Audit Overview” section. General overview is presented in AS-IS section and all found issues can be found in Audit overview section.

We didn’t found any high or critical vulnerabilities in the smart contracts but three low ones and also outlined six concerns, which may not have any security effect, but should be presented in the report.

## Severity Definitions

| Risk Level | Description |
|---|---|
| Critical | Critical vulnerabilities are usually straightforward to exploit and can lead to tokens lose etc. |
| High | High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions |
| Medium | Medium-level vulnerabilities are important to fix; however, they can’t lead to tokens lose |
| Low | Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can’t have significant impact on execution |
| Lowest / Code Style / Info | Lowest-level vulnerabilities, code style violations and info statements can’t affect smart contract execution and can be ignored. |

HackCtrl

# AS-IS Overview

## Intended Functionality

The submitted code implements NFT staking pool, NFT contract and upgradeable contracts with proxy mechanism.

### Upgradeability

By design, smart contracts are immutable. The basic idea is using a proxy for upgrades. The first contract is a simple wrapper or "proxy" which users interact with directly and is in charge of forwarding transactions to and from the second contract, which contains the logic.

There are 5 smart contracts aims to provide proxy pattern: main ProxyRegistry contract which is inherited from OwnableDelegateProxy, OwnedUpgradeabilityStorage, OwnedUpgradeabilityProxy, ProxyRegistry. Most of actions operated by proxy owner.

Most of proxy pattern code borrowed from open-source project Open Sea. It's a code running in a production, that is why it also approved by real live interactions.

One minor changes it's a fake ETH receiver in Proxy, which only generate a log event.

### NFT

EddaNft collect limited edition NFTs from Edda. There is ERC1155 standard interface for contracts that manage multiple token types. ERC1155 contains the logic that whitelists an operator address. Also, it extended by Mint, Burn, Tradable, Pause abilities and operated by appropriate roles like Minter, Pausable and WhitelistAdmin.

Most of multi-token standard code and role management also borrowed from open-source production implementations.

HackCtrl

## NFT Staking

Core contract for Edda NFT staking, inherited from PoolTokeWrapper.
It contains important actions:

- create pool;
- create and update card;
- stake;
- withdraw;
- transfer;
- redeem;
- rescue points;
- set artist, controller, rescuer;
- set controller share;
- withdraw fee
- update card points

There are also the most sensitive calculations here - this is where earnings and commissions are calculated.

For each account earning lives in updateReward modifier. It assigned to stake, withdraw, redeem and rescue points functions. How many earned points is calculating by formula:

$$\frac{PoolBalanceOfAccount * (BlockTime - AccountLastUpdateTime) * PoolRewardRate}{1e18} + AccountPoolPoints$$

Fee calculation lives in redeem function if mint fee appropriate card is more than 0.

$$ControllerShare = \frac{SentETH * PoolControllerShare}{1000}$$

$$ArtistRoyalty = SentETH - ControllerShare$$

All formulas were taken from the code, any additional specification was not provided. We cannot be responsible for the correctness of the project business logic and staking limitations. Also, we cannot predict interaction with other protocols and unknown smart contracts.

## Smart Contracts Audited

The following files have been covered in the audit process:

contracts
├── EddaNft.sol
├── EddaNftStake.sol

lib contracts
├── ERC1155.sol
├── ERC1155Metadata.sol
├── ERC1155MintBurn.sol
├── ERC1155Tradable.sol
├── IERC165.sol
├── IERC1155.sol
├── IERC1155TokenReceiver.sol
├── MinterRole.sol
├── OwnableDelegateProxy.sol
├── OwnedUpgradeabilityProxy.sol
├── OwnedUpgradeabilityStorage.sol
├── Pausable.sol
├── PauserRole.sol
├── PoolTokenWrapper.sol
├── Proxy.sol
├── ProxyRegistry.sol
├── Roles.sol
├── Strings.sol
├── WhitelistAdminRole.sol

openzeppelin contracts
├── Context.sol
├── Ownable.sol
├── SafeMath.sol
├── Address.sol

# Audit Overview

## Critical

No critical vulnerabilities were found.

## High

No high vulnerabilities were found.

## Medium

No medium vulnerabilities were found.

## Low

1. Exceed max supply

In ERC1155Tradable contract (line 119) mint require not exceed maxSupply, but before minting and compare current supply with max supply. Minter Role available to mint more tokens than max supply with any quantity in params. It will affect updateTokenMaxSupply and it will never work, because current supply will exceed max supply in line 175.

2. Timestamp manipulations

In EddaNftStaking contract (94 line) points earning based on block timestamp. Be aware that block.timestamp can be manipulated by miners, but with the following constraints: it cannot be stamped with an earlier time than its parent and it cannot be too far in the future.

3. Safe transfer from account

In PoolTokenWrapper contract (line 43) it should use safeTransferFrom from openzeppelin SafeERC20 library instead of plain transferFrom.

## Lowest / Code Style

1. Code duplication

In ERC1155Metadata contract (76 line) _uint2str function duplicate the same function from utils Strings library (51 line)

## Concerns

1. Reentrancy

Prevent reentrancy by adding Reentrancy Guard to all public and external functions. This contract module helps prevent reentrant calls to a function. Inheriting from ReentrancyGuard will make the nonReentrant modifier available, which can be applied to functions to make sure there are no nested (reentrant) calls to them

2. Proxy usage

In Solidity, the code that is inside a constructor or part of a global variable declaration is not part of a deployed contract's runtime bytecode. This code is executed only once, when the contract instance is deployed. As a consequence of this, the code within a logic contract's constructor will never be executed in the context of the proxy's state. To rephrase, proxies are completely oblivious to the existence of constructors. It's simply as if they weren't there for the proxy.

The problem is easily solved though. Logic contracts should move the code within the constructor to a regular initializer function, and have this function be called whenever the proxy links to this logic contract. Special care needs to be taken with this initializer function so that it can only be called once, which is one of the properties of constructors in general programming.

This is why when we create a proxy, you can provide the name of the initializer function and pass parameters. It is necessary to make sure that the initialized function can only be called once.

During proxy registration in ProxyRegistry contract (70 line), OwnableDelegateProxy takes one of the param like calldata. It's the initialize function with two params. We do not thoroughly know which contracts will be upgradeable, because upgradeability pattern does not connect with that contract directly and deploy separately, but no one contracts contain initialize function instead of constructor and all of those also contain constructor, which is not supported by upgradeable pattern. Please be careful in this case, also considering that this logic is not covered by tests at all.

3. Suspicion of incorrect logic when setting an artist or controller

In EddaNftStake contract (208 and 217 lines) please pay attention to those implementations. First 3 lines do nothing in result - save amount, burn it, then add it again. So pendingWithdrawals of artist is not changed. The same for controller.

4. Pool creation

In EddaNftStake contract (246 line) please pay attention if implementation is not assume to override existing pool, the check of only rewardRate is not enough. In this case id must be checked instead.

5. Controller share

Be careful during setting up a pool controller share, because multiplication of received ETH in wei and controller share value must be strict more than 1000. Please re-check the formula from above overview.

6. Points reward

In EddaNftStake contract be careful during stake invocation - if the numerator with the product of three factors (point earning formula from overview) is less than 1e18, it should be reverted every time. Please make sure of a minimum indicator of rewardRate. Also, none of the factors should be equal to zero, check carefully initial stake action.


# Conclusion

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. For the contract, high level description of functionality was presented in As-is overview section of the report.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Project has tests for the business logic and blockchain interaction. Nevertheless, test coverage is very small, many cases are completely absent.

The overall quality of the code submitted for the audit is good.
Best practice recommendations have largely been followed. Existing, audited code has been used whenever possible in the form of the OpenZeppelin libraries (https://openzeppelin.com/). A safe math library has been used for arithmetic operations to avoid overflow and underflow issues. Code layout mostly follows the official Solidity style guide

https://docs.soliditylang.org/en/v0.6.6/style-guide.html

# Disclaimers

## HackControl Disclaimer

The smart contracts given for audit have been analyzed in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report; the Source Code functionality (performing the intended functions).

The audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

## Technical Disclaimer

Smart contracts are deployed and executed on block chain platform. The platform, its programming language, and other software related to the smart contract can have own vulnerabilities that can lead to hacks. Thus, the audit can’t guarantee explicit security of the audited smart contracts.