SMART CONTRACT AUDIT

Report for:	EDDASwap
Date:	26.08.2021 - 02.09.2021

Table of Contents

Executive Summary

Hackcontrol (Provider) was contracted by **EDDASwap** (Client) to carry out a smart contract audit.

The first audit was conducted between **02.04.2021 - 13.04.2021**.

The second audit was conducted between **24.06.2021 - 25.06.2021**.

The third audit was conducted between **26.08.2021 - 02.09.2021**.

The objectives of the audit are the following:

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 behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

Secondary audit

The objective of the secondary audit is to verify the security of the updated functionality and determine the correctness of new changes.

The secondary audit has found the issue in the updated functionality connected to the funds transfer. Nevertheless, the issue was successfully fixed.

Third audit

The objective of the third audit is to:

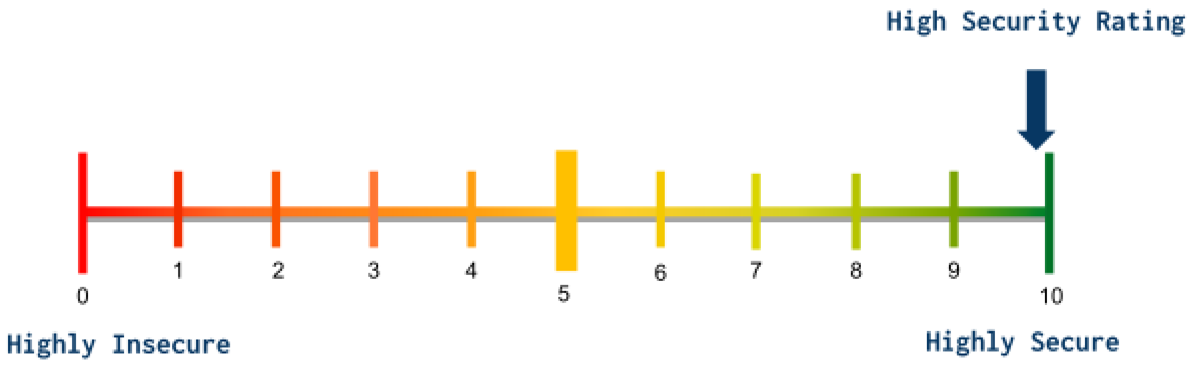
- determine the difference between the last version of the code and the previously re-audited one;

- evaluate the impact of newly added functionality on the previously audited one;

- verify the security of the updated functionality and determine the correctness of new changes.

The third audit has found several issues in the updated functionality related to the code quality, to the incorrect initialization and missing checks.

According to our research, after performing the audit, the security rating of the client's smart contract is **Medium**.



Scope

The Smart contract source code was taken from the Client's repository.

Repository - https://github.com/EddaSwap/eddaswap-core

Commit id - 527617179d15a5a611ed2d0020da991752ffd4a0

Secondary audit commit id - 0b46efa33647e9601e8bbe776728257cf2a85622

Third audit commit id - 70c9fb2258045086e7e1137b0bd8b3316bf443db

The last reviewed commit id - 6a5633d12357c39f10cca2ec85ccae66791742f8

The following list of the information systems was in scope of the audit.

#	Name
1.	EddaERC20.sol
2.	EddaFactory.sol
3.	EddaPair.sol

Project Tree

The following files have been checked within the audit process:

```
contracts
├── EddaERC20.sol
├── EddaFactory.sol
├── EddaPair.sol


library contracts
├── Math.sol
├── SafeMath.sol
├── UQ112x112.sol


interfaces
├── IEddaCallee.sol
├── IEddaERC20.sol
├── IEddaFactory.sol
├── IEddaPair.sol
├── IERC20.sol
```

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.
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 and remediate recommendation writing

Severity Definition

The level of criticality of each vulnerability is determined based on the potential impact of loss from successful exploitation as well as ease of exploitation, existence of exploits in public access and other factors.

Severity	Description
High 🟥🟥🟥🟥	High-level vulnerabilities are easy to exploit and may provide an attacker with full control of the affected systems, also may lead to significant data loss or downtime. There are exploits or PoC available in public access.
Medium 🟨🟨🟨	Medium-level vulnerabilities are much harder to exploit and may not provide the same access to affected systems. No exploits or PoCs are available in public access. Exploitation provides only very limited access.
Low 🟩🟩	Low-level vulnerabilities exploitation is extremely difficult, or impact is minimal.
Info 🟦	Information-level vulnerabilities provide an attacker with information that may assist them in conducting subsequent attacks against target information systems or against other information systems, which belong to an organization.

Summary of Findings

The table below shows the vulnerabilities and their severity. A total of four (8) vulnerabilities were found.

Title	Severity
Missing checks for the treasury address (resolved)	Medium
Different Solidity versions	Medium
Solidity version update	Medium
Missing check for the router address	Low
Overpowered role	Info
Unused contract	Info
Incorrect documentation	Info
“Magic numbers”	Info

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.

Key Findings

■■■ Missing check for the treasury address

#1	Description
	No checks for treasury to be set.

Evidences

EddaFactory.sol, constructor(), setTreasury()
EddaPair.sol, initialize(), swap() - line 174, 178

The treasury address is set in the factory contract. However, the treasury is not set in the constructor but in a separate method. Thus, there is a scenario, where a pair can be created with zero treasury. This is the first part of the issue.

The pair construct does not check the address of the treasury against a zero address (neither in the initializer, nor in the swap()). Thus, there is a scenario, where the swap fee is sent to the zero address and locked forever. That is the second part of the issue. Also there is no way to set the treasury in the pair.

Also, the EddaFactory.setTreasury() allows setting zero addresses to the treasury, which also affects pair creation.

The issue is marked as medium for several reasons: there are several scenarios, where fees will be locked forever on zero address (which already causes the medium risk to be set), there is no ability for the pair to avoid it (neither by checks, nor by the additional setter), there are no standard zero address checks.

Recommendations

Prevent zero address setting in the EddaPair initializer OR
Prevent fee subtraction in case of zero treasury OR
Verify that the treasury is always set by adding it to the factory constructor along with a zero check in the factory treasury setter.

The auditor team highly recommends implementing complex security measures for this issue

Different Solidity versions

#2	Description
Different pragma directives are used.	

Evidences

Throughout the project (including interfaces). Version used: '=0.5.16', '>=0.5.0'
Issue is classified as Medium, because it is included in the list of standard smart contracts' vulnerabilities.

Recommendations

Use the same pragma directives for the entire project.

Solidity version update

#3	Description
The solidity version should be updated.	

Evidences

Throughout the project (including interfaces).
Issue is classified as Medium, because it is included in the list of standard smart contracts' vulnerabilities.

Recommendations

You need to update the solidity version to the latest one - at least to 0.6.12, though 0.7.6 will be the best option. This will help to get rid of bugs in the older versions.

■■ Missing check for the router address

#4	Description
	No checks for the router to be set.

Evidences

EddaFactory.sol, constructor(), setRouter()
EddaPair.sol, initialize(), swap() - line 67, 161

The router address is set in the factory contract. However, there is a scenario, where a pair can be created with zero routers.

The pair does not check the address of the router against zero address (neither in the initializer, nor in the swap()). Thus, there is a scenario where the swap cannot be performed, since the router address is set to 0. Also there is no way to set the treasury in the pair.

Also, the EddaFactory.setTreasury() allows setting a zero address to the router, which also affects pair creation.

The issue is marked as low for several reasons: there are several scenarios, where the pair is created without the ability to swap tokens and there are no standard zero address checks. Nevertheless there are no issues with funds - they can be added or removed from the pair.

Recommendations

Prevent zero address setting in the EddaPair initializer

■ Overpowered role

#5	Description
	The FeeToSetter address has unique permissions to change the setter and collect the fees. Though there is no way to change that address

Evidences

Factory.sol

Recommendations

Consider usage of Ownable pattern or Role model from OpenZeppelin' AccessControl contract

Unused contract

#6	Description
Standard Migration contract does not have any role in the system and can be removed	
Evidences	
Migration.sol	
Recommendations	
Remove the unused standard contract	

Incorrect documentation

#7	Description
Doc-comment to the function _mintFee(), line 93 states incorrect multiplier: 6 instead of 5	
Evidences	
EddaPair.sol	
Recommendations	
Correct the docstring	

“Magic numbers”

#8	Description
EddaPair.sol: _mintFee(), line 102 and swap(), line 187 contain “magic” numbers which are not well documented and which were changed several times. It is better to move them to constants in order to increase readability and secure future development	

Evidences
EddaPair.sol
Recommendations
Use named constants instead of “magic” numbers.

Appendix A. Automated Tools

Scope	Tools Used
Smart-contracts Security	Mythril Solhint Slither Smartdec