SMART CONTRACT AUDIT

Report for:	EDDA-master
Date:	26.08.2021 - 02.09.2021
Reaudit Date:	03.10.2021 - 05.10.2021

HackCtrl

Table of Contents

Executive Summary

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

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

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

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

The fourth audit (re-audit) was conducted during **03.10.2021-04.10.2021**

Objectives of the audits 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.

Second 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 fees calculation. The issue was particularly resolved and the initial functionality was verified. It requires further testing for edge-cases and potential re-audit.

Third audit

The objective of the third audit is to:

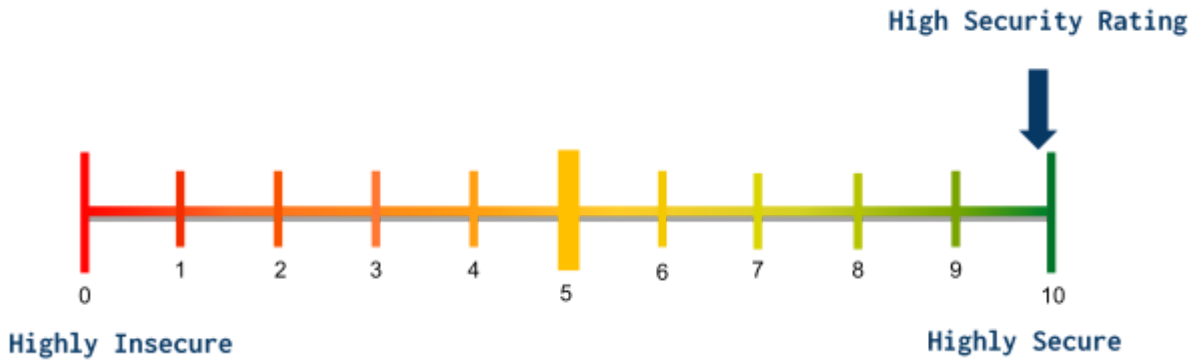
- determine the difference between the last version of the code and the previously re-audited one;

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

- verify 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 incorrect fee calculations. Also, some functionality needs clarification and confirmation from the Client's end.
After the fourth audit (re-audit), the Client has verified all unclear functionality and corrected critical issues.

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



Scope

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

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

Commit id - 156d1250359e02ef3423236d1d5b6511bf15df29

Secondary audit commit id - ee100ebe8b5b038672e46fc72b092233114aef85

Third audit commit id - 2d6a62321c4066e2b4d2a0dc7a969ff2592fc328

Fixed code (after the third audit) commit id - 29dd068730d62d0e887819833c19d45abb0bd192

The fourth audit (re-audit) commit id - fbeaf66eb02b7f3144326f864e9b381cc67f1fff

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

#	Name
1.	EddaMigrator.sol
2.	EddaRouter.sol

Project Tree

The following files have been checked within the audit process:

```
contracts
├── EddaMigrator.sol
├── EddaRouter.sol


interfaces
├── IEddaMigrator.sol
├── IEddaRouter01.sol
├── IEddaRouter02.sol
├── IERC20.sol
├── IWETH.sol


interfaces/V1
├── IUniswapV1Exchange.sol
├── IUniswapV1Factory.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

HackCtrl

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 for 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.

HackCtrl

Summary of Findings

The table below shows the vulnerabilities and their severity. A total of 6 vulnerabilities were found.

Title	Severity
Inconsistency in calculations (resolved - verified by customer)	**High**
Fee withdrawal can be omitted (resolved)	**High**
Incorrect calculations (resolved)	Medium
Different Solidity versions	Medium
Solidity version update	Medium
Unused contract	Info

The overall quality of the code submitted for the audit is very good.

Best practice recommendations have largely been followed. Existing, the 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

■■■■ Inconsistency in calculations

#1	Description

Inconsistency in commission calculation

Evidences

EddaRouter.sol, swapTokensForExactETH(), line 624

At line 632 we get input amounts calculated from the amountOut parameter. After that we calculate a withdrawal fee from the input amount of the token (line 634), so the fee is transferred from the user to the treasury.

After that, the full input amount is transferred from the user and goes to swap (line 638). Thus, the user should approve (amountIn + fee) for the input token for this operation. So, the fee should come over the input amount

On the other hand, in other swap functions the user approves only the input amount of the token, and the fee is withdrawn from that amount.

Thus, we have 2 different mechanics for the fee withdrawal (one - over the input, and other - from the input).

Also, the same applies for the swapTokensForExactETH() and swapETHForExactTokens() functions.

Recommendations

Verify that the functionality works as expected, since inconsistency in calculations may lead to incorrect calculations on the frontend, thus - to incorrect amounts sent by the user. Or provide the fix to have consistent fee mechanics.

Post-audit: functionality verified by the customer

■■■■ Fee withdrawal can be omitted

#2	Description

Users can provide a swap without paying any fee.

Evidences

All newly added functions for swaps with support of an internal token fee do not withdraw commission to the treasury. Thus, swaps can be performed for regular tokens without paying a treasury fee.

Recommendations

Verify that the functionality works as expected. Provide testing scenarios with fee calculation for the treasury in case of swaps with support of internal token fees.

Post-audit: functions deleted

■■■ Incorrect calculations

#3	Description

Commission is performed twice.

Evidences

EddaRouter.sol, swapExactTokensForETH(), line 561
EddaRouter.sol, swapTokensForExactETH(), line 543

At line 561 we get output amounts (EddaLibrary.getAmountsOut()). Each output amount is already decreased by 0.1% (999/1000) in EddaLibrary.getAmountOut(). Nevertheless, amountETH has the second fee performed and now becomes not the <expected ETH> * 99.9% but <expected ETH> * 99.9% * 99.9%

The same applies to swapTokensForExactETH() - the function receives the correct final ETH value, and EddaLibrary.getAmountsIn() calculates and returns the input amount already increased by the 0.1% commission. So, a double fee is performed.

Recommendations
Verify the functionality and correct the calculation

Different Solidity versions

#4	Description
Different pragma directives are used.	

Evidences
Throughout the project (including interfaces). Version used: '=0.6.6', '>=0.5.0', '>=0.6.0', '>=0.6.2' 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

#5	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 (0.6.12 - the best variant). This will help to get rid of bugs in the older versions.

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

Appendix A. Automated Tools

Scope	Tools Used
Smart-contracts Security	Mythril Solhint Slither Smartdec