QuillAudits

Audit Report
September, 2021

For

euler.tools

Contents

Scope of the Audit

The scope of this audit was to analyze and document the Euler Staking smart contract codebase for quality, security, and correctness.

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis
In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis
Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption
In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Mythril, Slither, SmartCheck, Surya, Solhint.

Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

Risk-level	Description
High	A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.
Medium	The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.
Low	Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.
Informational	These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

Type	High	Medium	Low	Informational
Open	0	0	0	0
Acknowledged	0	1	0	0
Closed	0	2	3	8

Introduction

During the period of **September 02, 2021 to September 06, 2021** - QuillAudits Team performed a security audit for Euler Staking smart contracts.

The code for the audit was taken from following the official link: https://github.com/eulertools/contracts/blob/c3fa6a76387024916cdb8f645f686c841aea6464/contracts/EulerStaking.sol

Note	Date	Link
Version 1	September	21f1bbc712381bb9dfb19da1bb3c6369dc372307
Version 2	September	99b273b3a276f4215bc7247c26c5e23f8f010729
Version 3	September	c3fa6a76387024916cdb8f645f686c841aea6464

Deployed code: https://bscscan.com/address/0xb18fab4c6f054e734ea169561787cc87928f54ee#code

Issues Found

High severity issues

No issues were found.

Medium severity issues

1. EulerPerBlock miscalculation after setEulerPerBlock()

Line	Code

266-269	function setEulerPerBlock(uint256 _eulerPerBlock) external onlyOwner {
	require(_eulerPerBlock > 0, "EULER per block should be greater than 0!");
	eulerPerBlock = _eulerPerBlock;
	}

Description
The eulerPerBlock variable is used for calculations of rewards. So if this value is modified by calling the setEulerPerBlock() function, without updating the pending rewards first, then the pool rewards calculation will be incorrect.

Remediation
We recommend calling the updatePool(0) before modifying eulerPerBlock.

Status: Closed
In Version 2, updatePool() was called before modifying eulerPerBlock.

2. User's Lockdown period resets after withdraw()

Line	Code
221	user.lastClaim = block.timestamp; emit Withdraw(msg.sender, pid, amount);

Description
The lockdown period of 30 days should reset whenever a user deposits or claims their rewards. But in the withdraw() function, the user's lockdown period is again reset, which is not according to the specification.

Remediation
We recommend removing the statement at line 221, which resets the user's lastClaim variable.

Status: Closed
In version 2, the statement to reset the lastClaim was removed.

3. Centralization Risks

Description
The role owner has the authority to :
- Change lockdown period's duration
- Change the Euler per block reward rate
- Set minimum deposit amount
- Set maximum deposit amount
- Change the transaction fees

It has to be noted that the owner can change the eulerTxFee at any time. The eulerTxFee variable is used to take into consideration the transaction fee charged by the Euler Token contract that goes to the burn address. Initially, it is set to 100 which is equal to the 1% charged by the token contract.

If it is increased, then some of that fee will go to the staking contract. If it is decreased to a number below 100, then it can be a loss for the staking contract.

Remediation
We advise the client to handle the governance account carefully to avoid any potential hack. We also advise the client to consider the following solutions:

1. Time-lock with reasonable latency for community awareness on privileged operations;
2. Multisig with community-voted 3rd-party independent co-signers;
3. DAO or Governance module increasing transparency and community involvement;

Status: Acknowledged by the Auditee

Low level severity issues

4. poolInfo[] stores only one element

Line	Code
43	PoolInfo[] public poolInfo;

Description
The addPool() function adds new elements into the poolInfo array. As we can see, there is only one pool added. The array structure can be simplified into a single struct variable.

Remediation
We recommend declaring poolInfo as a struct variable instead of an array. After that there will be no need to pass pid to the deposit, withdraw and claim function as parameters. This can make the transaction cheaper in terms of gas.

Status: Closed
In version 2, poolInfo was changed to a variable instead of an array.

5. Missing zero address validation

Line	Code

43	function setEulerToken(IERC20 _euler) external onlyOwner {
	require(address(euler) == address(0), "Token already set!");
	euler = _euler;
	addPool(10, 30 days);
	}

Description
When setting the Euler token address, it should be checked for zero address. Otherwise, tokens sent to the zero address may be burnt forever.

Remediation
Use a require statement to check for zero address when setting the Euler token address.

Status: Closed
In version 2, _euler was checked for zero address.

6. Staking can start without adding pool

Line	Code

77-81	function startStaking(uint256 startBlock) external onlyOwner {
	require(poolInfo.lastRewardBlock == 0, "Staking already started");
	poolInfo.lastRewardBlock = startBlock;
	emit StartStaking(msg.sender, startBlock);
	}

Description
In the function startStaking(), the require statement checks if there is an existing pool and the staking has not started. But after the contract's deployment, it is always going to be true. There needs to be a better way to check if a pool has been added and staking has not started.

Remediation
We recommend that you change the require statement to this:

```
require(
    poolInfo.lockupDuration == 30 days && poolInfo.lastRewardBlock
== 0,
    "Staking already started"
);
```

Status: Closed
In version 3, the team implemented our recommended solution.

Informational

7. Redundant Code

Line	Code
45	uint256 public totalAllocPoint = 10;
99-100	.mul(pool.allocPoint) .div(totalAllocPoint);

Description
pool.allocPoint and totalAllocPoint always remain the same, i.e. 10. So multiplying and dividing with it will always result in the same value and will only waste gas.

Remediation
We recommend removing all the instances of totalAllocPoint and pool.allocPoint from the contract logic.

Status: Closed
In version 2, all the logic related to allocation points was removed.

8. Incorrect versions of Solidity

pragma solidity >=0.8.0 <=0.9.0

Description
solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statements.

Remediation
Use a simple pragma version. Consider using the latest version of Solidity for testing.

Status: Closed
In version 2, the pragma version was locked to 0.8.7 .

9. Missing Events for Significant Transactions

Description
The missing event makes it difficult to track off-chain liquidity fee changes. An event should be emitted for significant transactions calling the following functions:

setEulerPerBlock
setMinDepositAmount
setMaxDepositAmount
setEulerTxFee
setLockupDuration
startStaking

Remediation
We recommend emitting an event to log the update of the important variables.

Status: Closed
In version 2, appropriate events were emitted for the update of important variables.

10. State variables that could be declared constant

maxFee
totalAllocPoint

Description
The above constant state variables should be declared constant to save gas.

Remediation
Add the constant attributes to state variables that never change.

Status: Closed
In version 2, the state variables mentioned were declared constant.

11. Change memory to storage

Line	Code
258	PoolInfo memory pool = poolInfo[pid];

Description
When a storage variable is created in function locally by look up, it simply references data already allocated on Storage. No new storage is created. More info here.

Remediation
Use the storage attribute for the variable pool.

Status: Closed
In version 2, poolInfo was declared as a single variable.

12. Public function that could be declared external

Description
The following public functions that are never called by the contract should be declared external to save gas:
- claim

Remediation
Use the external attribute for functions never called from the contract.

Status: Closed
In version 2, the claim function was declared external.

13. Explicitly check for pool pid validity

Description
There is no explicit check to validate if a pool exists according to the given pid. The current implementation will revert or give out errors based on the compiler's check for array length bounds.

Remediation
A better method will be to have a modifier that validates a pool by its provided pid, before trying to access the pool in the different functions like deposit, withdraw, claim, etc.

Status: Closed
In version 2, poolInfo was declared as a single variable. So passing a pid parameter was no longer required.

14. Wrong error message

Line	Code
256	require(amount > 0, "Withdrawing more than 0!");

Description
The error message when the user tries to withdraw amount zero is incorrect.

Remediation
We recommend that it should state the error clearly that the user is trying to withdraw zero amount.

Status: Closed
In version 3, the error message was updated.

Functional Tests

Function Names	Testing results
setEulerToken()	Passed
startStaking()	Passed
setLockupDuration()	Passed
getUserInfo()	Passed
deposit()	Passed
withdraw()	Passed
withdrawAll()	Passed
claim()	Passed
setEulerPerBlock()	Passed
setMinDepositAmount()	Passed
setMaxDepositAmount()	Passed
setEulerTxFee()	Passed
getTotalUsers	Passed
getUserInfoByPosition	Passed

Automated Testing

Slither

```
INFO:Detectors:
Reentrancy in EulerStaking.claim() (EulerStaking.sol#287-305):
        External calls:
        - claimedAmount = safeEulerTransfer(msg.sender,user.pendingRewards) (EulerStaking.sol#295-298)
                - returndata = address(token).functionCall(data,SafeERC20: low-level call failed) (@openzeppelin\contracts\token\ERC20\
utils\SafeERC20.sol#92)
                - (success,returndata) = target.call{value: value}(data) (@openzeppelin\contracts\utils\Address.sol#131)
                - euler.safeTransfer(to,amount) (EulerStaking.sol#335)
        External calls sending eth:
        - claimedAmount = safeEulerTransfer(msg.sender,user.pendingRewards) (EulerStaking.sol#295-298)
                - (success,returndata) = target.call{value: value}(data) (@openzeppelin\contracts\utils\Address.sol#131)
        State variables written after the call(s):
        - poolInfo.rewardsAmount = poolInfo.rewardsAmount.sub(claimedAmount) (EulerStaking.sol#302)
        - user.pendingRewards = user.pendingRewards.sub(claimedAmount) (EulerStaking.sol#300)
        - user.lastClaim = block.timestamp (EulerStaking.sol#301)
        - user.rewardDebt = user.amount.mul(poolInfo.accEulerPerShare).div(1e12) (EulerStaking.sol#304)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities
INFO:Detectors:
EulerStaking.updatePool() (EulerStaking.sol#187-210) uses a dangerous strict equality:
        - poolInfo.depositedAmount == 0 (EulerStaking.sol#198)
EulerStaking.withdraw(uint256) (EulerStaking.sol#250-279) uses a dangerous strict equality:
        - user.amount == 0 (EulerStaking.sol#273)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:

Reentrancy in EulerStaking.deposit(uint256) (EulerStaking.sol#212-248):
        External calls:
        - euler.safeTransferFrom(address(msg.sender),address(this),amount) (EulerStaking.sol#233)
        State variables written after the call(s):
        - poolInfo.depositedAmount = poolInfo.depositedAmount.add(amount) (EulerStaking.sol#237)
        - user.amount = user.amount.add(amount) (EulerStaking.sol#236)
        - user.rewardDebt = user.amount.mul(poolInfo.accEulerPerShare).div(1e12) (EulerStaking.sol#239)
        - user.lastClaim = block.timestamp (EulerStaking.sol#240)
        - user.exists = true (EulerStaking.sol#244)
Reentrancy in EulerStaking.withdraw(uint256) (EulerStaking.sol#250-279):
        External calls:
        - euler.safeTransfer(address(msg.sender),amount) (EulerStaking.sol#266)
        State variables written after the call(s):
        - poolInfo.depositedAmount = poolInfo.depositedAmount.sub(amount) (EulerStaking.sol#268)
        - user.amount = user.amount.sub(amount) (EulerStaking.sol#267)
        - user.rewardDebt = user.amount.mul(poolInfo.accEulerPerShare).div(1e12) (EulerStaking.sol#270)
        - user.exists = false (EulerStaking.sol#275)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
EulerStaking.getUserInfo(address).userAux (EulerStaking.sol#151) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
Reentrancy in EulerStaking.deposit(uint256) (EulerStaking.sol#212-248):
        External calls:
        - euler.safeTransferFrom(address(msg.sender),address(this),amount) (EulerStaking.sol#233)
        State variables written after the call(s):
        - users.push(msg.sender) (EulerStaking.sol#243)
```

```
Reentrancy in EulerStaking.withdraw(uint256) (EulerStaking.sol#250-279):
        External calls:
        - euler.safeTransfer(address(msg.sender),amount) (EulerStaking.sol#266)
        State variables written after the call(s):
        - removeUser(msg.sender) (EulerStaking.sol#277)
                - users[i] = users[i + 1] (EulerStaking.sol#313)
                - users.pop() (EulerStaking.sol#316)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in EulerStaking.claim() (EulerStaking.sol#287-305):
        External calls:
        - claimedAmount = safeEulerTransfer(msg.sender,user.pendingRewards) (EulerStaking.sol#295-298)
                - returndata = address(token).functionCall(data,SafeERC20: low-level call failed) (@openzeppelin\contracts\token\ERC20\
utils\SafeERC20.sol#92)
                - (success,returndata) = target.call{value: value}(data) (@openzeppelin\contracts\utils\Address.sol#131)
                - euler.safeTransfer(to,amount) (EulerStaking.sol#335)
        External calls sending eth:
        - claimedAmount = safeEulerTransfer(msg.sender,user.pendingRewards) (EulerStaking.sol#295-298)
                - (success,returndata) = target.call{value: value}(data) (@openzeppelin\contracts\utils\Address.sol#131)
        Event emitted after the call(s):
        - Claim(msg.sender,claimedAmount) (EulerStaking.sol#299)
Reentrancy in EulerStaking.deposit(uint256) (EulerStaking.sol#212-248):
        External calls:
        - euler.safeTransferFrom(address(msg.sender),address(this),amount) (EulerStaking.sol#233)
        Event emitted after the call(s):
        - Deposit(msg.sender,amount) (EulerStaking.sol#247)
```

```
Reentrancy in EulerStaking.withdraw(uint256) (EulerStaking.sol#250-279):
        External calls:
        - euler.safeTransfer(address(msg.sender),amount) (EulerStaking.sol#266)
        Event emitted after the call(s):
        - Withdraw(msg.sender,amount) (EulerStaking.sol#271)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
EulerStaking.withdraw(uint256) (EulerStaking.sol#250-279) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp > user.lastClaim + poolInfo.lockupDuration,You cannot withdraw yet!) (EulerStaking.sol#2
52-255)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Address.isContract(address) (@openzeppelin\contracts\utils\Address.sol#26-36) uses assembly
        - INLINE ASM (@openzeppelin\contracts\utils\Address.sol#32-34)
Address._verifyCallResult(bool,bytes,string) (@openzeppelin\contracts\utils\Address.sol#189-209) uses assembly
        - INLINE ASM (@openzeppelin\contracts\utils\Address.sol#201-204)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Different versions of Solidity is used:
        - Version used: ['0.8.7', '>=0.4.22<0.9.0', '^0.8.0']
        - ^0.8.0 (@openzeppelin\contracts\utils\Address.sol#3)
        - ^0.8.0 (@openzeppelin\contracts\utils\Context.sol#3)
        - 0.8.7 (EulerStaking.sol#2)
        - ^0.8.0 (@openzeppelin\contracts\token\ERC20\IERC20.sol#3)
        - >=0.4.22<0.9.0 (Migrations.sol#2)
```

```
        - ^0.8.0 (@openzeppelin\contracts\access\Ownable.sol#3)
        - ^0.8.0 (@openzeppelin\contracts\token\ERC20\utils\SafeERC20.sol#3)
        - ^0.8.0 (@openzeppelin\contracts\utils\math\SafeMath.sol#3)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
INFO:Detectors:
Pragma version^0.8.0 (@openzeppelin\contracts\utils\Address.sol#3) necessitates a version too recent to be trusted. Consider deploying
with 0.6.12/0.7.6
Pragma version^0.8.0 (@openzeppelin\contracts\utils\Context.sol#3) necessitates a version too recent to be trusted. Consider deploying
with 0.6.12/0.7.6
Pragma version0.8.7 (EulerStaking.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (@openzeppelin\contracts\token\ERC20\IERC20.sol#3) necessitates a version too recent to be trusted. Consider deplo
ying with 0.6.12/0.7.6
Pragma version>=0.4.22<0.9.0 (Migrations.sol#2) is too complex
Pragma version^0.8.0 (@openzeppelin\contracts\access\Ownable.sol#3) necessitates a version too recent to be trusted. Consider deploying
 with 0.6.12/0.7.6
Pragma version^0.8.0 (@openzeppelin\contracts\token\ERC20\utils\SafeERC20.sol#3) necessitates a version too recent to be trusted. Consi
der deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (@openzeppelin\contracts\utils\math\SafeMath.sol#3) necessitates a version too recent to be trusted. Consider depl
oying with 0.6.12/0.7.6
solc-0.8.7 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in Address.sendValue(address,uint256) (@openzeppelin\contracts\utils\Address.sol#54-59):
        - (success) = recipient.call{value: amount}() (@openzeppelin\contracts\utils\Address.sol#57)
Low level call in Address.functionCallWithValue(address,bytes,uint256,string) (@openzeppelin\contracts\utils\Address.sol#122-133):
        - (success,returndata) = target.call{value: value}(data) (@openzeppelin\contracts\utils\Address.sol#131)
```

```
INFO:Detectors:
Low level call in Address.sendValue(address,uint256) (@openzeppelin\contracts\utils\Address.sol#54-59):
        - (success) = recipient.call{value: amount}() (@openzeppelin\contracts\utils\Address.sol#57)
Low level call in Address.functionCallWithValue(address,bytes,uint256,string) (@openzeppelin\contracts\utils\Address.sol#122-133):
        - (success,returndata) = target.call{value: value}(data) (@openzeppelin\contracts\utils\Address.sol#131)
Low level call in Address.functionStaticCall(address,bytes,string) (@openzeppelin\contracts\utils\Address.sol#151-160):
        - (success,returndata) = target.staticcall(data) (@openzeppelin\contracts\utils\Address.sol#158)
Low level call in Address.functionDelegateCall(address,bytes,string) (@openzeppelin\contracts\utils\Address.sol#178-187):
        - (success,returndata) = target.delegatecall(data) (@openzeppelin\contracts\utils\Address.sol#185)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Parameter EulerStaking.addPool(uint256)._lockupDuration (EulerStaking.sol#59) is not in mixedCase
Parameter EulerStaking.setEulerToken(IERC20)._euler (EulerStaking.sol#69) is not in mixedCase
Parameter EulerStaking.setLockupDuration(uint256)._lockupDuration (EulerStaking.sol#83) is not in mixedCase
Parameter EulerStaking.setEulerPerBlock(uint256)._eulerPerBlock (EulerStaking.sol#88) is not in mixedCase
Parameter EulerStaking.setMinDepositAmount(uint256)._amount (EulerStaking.sol#96) is not in mixedCase
Parameter EulerStaking.setMaxDepositAmount(uint256)._amount (EulerStaking.sol#103) is not in mixedCase
Parameter EulerStaking.setEulerTxFee(uint256)._fee (EulerStaking.sol#113) is not in mixedCase
Parameter EulerStaking.pendingRewards(address)._user (EulerStaking.sol#120) is not in mixedCase
Parameter EulerStaking.getUserInfo(address)._user (EulerStaking.sol#144) is not in mixedCase
Constant EulerStaking.maxFee (EulerStaking.sol#36) is not in UPPER_CASE_WITH_UNDERSCORES
Variable Migrations.last_completed_migration (Migrations.sol#6) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
```

Result

No major issue was found. Some false positive errors were reported by the tool. All the other issues have been categorized above, according to their level of severity.

Closing Summary

Overall, smart contracts are very well written and adhere to guidelines.

No instances of Integer Overflow and Underflow vulnerabilities or Back-Door Entry were found in the contract, but relying on other contracts might cause Reentrancy Vulnerability.

Numerous issues were discovered during the initial audit. At the end, all of the issues were either fixed or acknowledged by the auditee.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the Euler platform. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Euler Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Audit Report September, 2021

For

euler.tools

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com