QuillAudits

Audit Report
August, 2021

SUNDIAL

Contents

Scope of Audit

The scope of this audit was to analyze and document the Sundial Token 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 analyzed 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.
SmartCheck.

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 security of smart contracts.

Code Review / Manual Analysis
Manual Analysis or review of code was done to identify new vulnerability or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of 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 possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Remix IDE, Truffle, Truffle Team, Ganache, Solhint, Mythril, Slither, SmartCheck.

Issue Categories

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

High severity issues

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 to be fixed before moving to a live environment.

Medium level severity issues

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 level severity issues

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 severity four issues which 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	1	1
Acknowledged	0	0	0	0
Closed	0	0	2	2

Introduction

During the period of **July 13, 2021 to July 21, 2021** - QuillAudits Team performed a security audit for Sundial smart contracts.

The code for the audit was taken from the following official link: https://github.com/TrySundial/Sundial/blob/master/contracts/DAISO.sol

Commit Hash: e884daa01d63c8d126b4782d05a853755a7745fd

Updated version commit: 0cb7f958f21cc4f3640662db673782b18a8df9e2

Issues Found – Code Review / Manual Testing

High severity issues

No issues were found

Medium severity issues

Low level severity issues

1. Potential use of "block.timestamp" as a source of randomness.

```
331
332         require(msg.sender != project.sender,"SENDER_SAME_PROJECT");
333         require(investSellDeposit > 0, "INVESTSELLDEPOSIT_IS_ZERO");
334         require(block.timestamp < cancelProjectForInvest.exitStopTime, "NOW_BIGGER_STOPTIME");
335
336         uint256 startTime;
337         if (block.timestamp <= project.startTime){
338             startTime = project.startTime;
339         } else {
340             startTime = block.timestamp;
341         }
342
```

Description

Contracts often need access to time values to perform certain types of functionality. Values such as block.timestamp, and block.number can give you a sense of the current time or a time delta, however, they are not safe to use for most purposes.

In the case of block.timestamp, developers often attempt to use it to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. Moreover, malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. However, miners can't set a timestamp smaller than the previous one (otherwise the block will be rejected), nor can they set the timestamp too far ahead in the future. Taking all of the above into consideration, developers can't rely on the preciseness of the provided timestamp.

Remediation
Developers should write smart contracts with the notion that block values are not precise, and the use of them can lead to unexpected effects. Alternatively, they may make use of oracles.

References
Safety: Timestamp dependence
Ethereum Smart Contract Best Practices - Timestamp Dependence
How do Ethereum mining nodes maintain a time consistent with the network?
Solidity: Timestamp dependency, is it possible to do safely?

Status: Open

2. Outdated Compiler Version (SWC 102)

```
1  pragma solidity 0.5.16;
2
3  import "../node_modules/@openzeppelin/contracts-ethereum-package/contracts/token/ERC20/IERC20.sol";
4  import "../node_modules/@openzeppelin/contracts-ethereum-package/contracts/utils/ReentrancyGuard.sol";
5
```

Description
Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

Remediation
It is recommended to use a recent version of the Solidity compiler which is Version 0.8.4

Status: Fixed

3. Public function that could be declared external

Description
A function with a public visibility modifier that is not called internally. Changing the visibility level to external increases code readability. Moreover, in many cases, functions with external visibility modifiers spend less gas compared to functions with public visibility modifiers.

The function definition in the file which are marked as public are below
"projectBalanceOf"
"deltaOf"
"investBalanceOf"
"createArbitrationForInvestor"

However, it is never directly called by another function in the same contract or in any of its descendants. Consider marking it as"external" instead.

Recommendations
Use the **external** visibility modifier for functions never called from the contract via internal call. Reading Link.

Status: Fixed

Informational

1. The Daiso contract has around 21 linting issues. The same issue was found and put in the automated section.

   Status: Open

2. Reentrancy Guard missing: Use the reentrancy OpenZeppelin guard to avoid improper Enforcement of Behavioral Workflow. Link1, Link2.

   Status: Fixed

3. The suggestion to use here would be an SPDX license. You can find a list of licenses here: https://spdx.org/licenses/

   The SPDX License List is an integral part of the SPDX Specification. The SPDX License List itself is a list of commonly found licenses and exceptions used in free and open or collaborative software, data, hardware, or documentation. The SPDX License List includes a standardized short identifier, the full name, the license text, and a canonical permanent URL for each license and exception.

   Status: Fixed

Functional test

Function Names	Testing results
createProject	Passed
projectBalanceOf	Passed
projectRefunds	Passed
withdrawFromProject	Passed
createStream	Passed
getStream	Passed
deltaOf	Passed
investBalanceOf	Passed
withdrawFromInvest	Passed
cancelInvest	Passed
createArbitrationForInvestor	Passed
getArbitration	Passed
createDisputeForProject	Passed
rule	Passed
submitEvidence	Passed
appeal	Passed
reclaimFunds	Passed
getCancelProjectForInvest	Passed

Automated Testing

Slither

Slither is a Solidity static analysis framework that runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses. After running Slither, we got the results below.

```
Reentrancy in DAISO.createDisputeForProject(uint256) (test1.sol#1073-1092):
        External calls:
        - arbitrations[projectId].disputeID = IArbitrator(arbitratorAddress).createDispute.value(msg.value)(2,) (test1.sol#1082)
        State variables written after the call(s):
        - arbitrations[projectId].status = Types.Status.Disputed (test1.sol#1083)
        - arbitrations[projectId].evidenceGroup = nextEvidenceGroup (test1.sol#1084)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities
INFO:Detectors:
OwnableWithoutRenounce._____gap (test1.sol#558) shadows:
        - Initializable._____gap (test1.sol#476)
PauserRoleWithoutRenounce._____gap (test1.sol#590) shadows:
        - Initializable._____gap (test1.sol#476)
ReentrancyGuard._____gap (test1.sol#713) shadows:
        - Initializable._____gap (test1.sol#476)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variable-shadowing
INFO:Detectors:
DAISO.createArbitrationForInvestor(uint256,string) (test1.sol#1088-1139) uses a dangerous strict equality:
        - require(bool,string)(arbitrations[projectId].reclaimedAt == 0,41) (test1.sol#1117)
DAISO.onlyInvest(uint256) (test1.sol#857-863) uses a dangerous strict equality:
        - require(bool,string)(msg.sender == streams[streamId].sender,1) (test1.sol#858-861)
DAISO.projectBalanceOf(uint256) (test1.sol#978-1004) uses a dangerous strict equality:
        - projectFundBalance == 0 (test1.sol#1000)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Reentrancy in DAISO.votingResult(uint256) (test1.sol#1121-1176):
        External calls:
        - require(bool,string)(IERC20(project.projectFundTokenAddress).transfer(project.sender,proposal.amount),26) (test1.sol#11
        State variables written after the call(s):
        - delete proposals[projectId] (test1.sol#1174)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
```

1. function createDisputeForProject:

   External calls:
   - arbitrations[projectId].disputeID = IArbitrator(arbitratorAddress).createDispute.value(msg.value)(2,)

   State variables written after the call(s):
   - arbitrations[projectId].status = Types.Status.Disputed
   - arbitrations[projectId].evidenceGroup = nextEvidenceGroup

   Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities

2. state-variable-shadowing:

OwnableWithoutRenounce._____gap shadows:
- Initializable._____gap

PauserRoleWithoutRenounce._____gap shadows:
- Initializable._____gap

ReentrancyGuard._____gap shadows:
- Initializable._____gap

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variable-shadowing

Solhint

Solhint is an open-source project created by https://protofire.io. Its goal is to provide a linting utility for Solidity code. Below are the results.

```
DAISO.sol
   22:2  error  Line length must be no more than 120 but current length is 123  max-line-length
  189:2  error  Line length must be no more than 120 but current length is 136  max-line-length
  201:2  error  Line length must be no more than 120 but current length is 152  max-line-length
  259:2  error  Line length must be no more than 120 but current length is 133  max-line-length
  346:2  error  Line length must be no more than 120 but current length is 121  max-line-length
  372:2  error  Line length must be no more than 120 but current length is 139  max-line-length
  375:2  error  Line length must be no more than 120 but current length is 143  max-line-length
  411:2  error  Line length must be no more than 120 but current length is 146  max-line-length
  482:2  error  Line length must be no more than 120 but current length is 123  max-line-length
  513:2  error  Line length must be no more than 120 but current length is 127  max-line-length
  516:2  error  Line length must be no more than 120 but current length is 160  max-line-length
  519:2  error  Line length must be no more than 120 but current length is 131  max-line-length
  521:2  error  Line length must be no more than 120 but current length is 131  max-line-length
  525:2  error  Line length must be no more than 120 but current length is 124  max-line-length
  530:2  error  Line length must be no more than 120 but current length is 124  max-line-length
  551:2  error  Line length must be no more than 120 but current length is 131  max-line-length
  553:2  error  Line length must be no more than 120 but current length is 131  max-line-length
  557:2  error  Line length must be no more than 120 but current length is 131  max-line-length
  714:2  error  Line length must be no more than 120 but current length is 123  max-line-length
  771:2  error  Line length must be no more than 120 but current length is 167  max-line-length
  847:2  error  Line length must be no more than 120 but current length is 130  max-line-length

✖ 21 problems (21 errors, 0 warnings)
```

Results

No major issues were 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. Some issues are still open after the review; It is recommended to fix them.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the Sundial 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 Sundial Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

SUNDIAL

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

audits@quillhash.com