Bloq

Smart Contracts

Security Assessment

February 15th, 2021

By:
Sheraz Arshad @ CertiK
sheraz.arshad@certik.org

Camden Smallwood @ CertiK
camden.smallwood@certik.org

Disclaimer

CertiK reports are not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security review.

CertiK Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

CertiK Reports should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

CertiK Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

What is a CertiK report?

- A document describing in detail an in depth analysis of a particular piece(s) of source code provided to CertiK by a Client.
- An organized collection of testing results, analysis and inferences made about the structure, implementation and overall best practices of a particular piece of source code.
- Representation that a Client of CertiK has indeed completed a round of auditing with the intention to increase the quality of the company/product's IT infrastructure and or source code.

Overview

Project Summary

Project Name	Bloq: Payment Splitter
Description	The audit contract comprise of payment splitting functionality where depsited ERC20 tokens and ether are distributed among payees on the basis of their share.
Platform	Ethereum; Solidity, Yul
Codebase	N/A
Commits	N/A

Audit Summary

Delivery Date	February 15th, 2021
Method of Audit	Static Analysis, Manual Review
Consultants Engaged	2
Timeline	February 13th, 2021 - February 15th, 2021

Vulnerability Summary

Total Issues	5
Total Critical	0
Total Major	0
Total Medium	0
Total Minor	0
Total Informational	5

Executive Summary

This report represents the results of CertiK's engagement with Bloq for the `PaymentSplitter` contract. All of the findings are informational and were remediated except one. The findings are related to gas optimizations, locking of compiler version and legibile naming of a function's name.

Files In Scope

ID	Contract	Location
PSR	PaymentSplitter.sol	PaymentSplitter.sol

Findings

ID	Title	Type	Severity	Resolved
PSR-01	Unlocked Compiler Version	Language Specific	Informational	✓
PSR-02	mappings can be packed in a struct	Gas Optimization	Informational	✓
PSR-03	Inefficient storage access	Gas Optimization	Informational	✓
PSR-04	Explicitly returning local variable	Gas Optimization	Informational	✓
PSR-05	Confusing function name	Coding Style	Informational	✓

PSR-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	PaymentSplitter.sol L3

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.0` the contract should contain the following line:

```
pragma solidity 0.6.0;
```

Alleviation:

Alleviations were applied as advised by locking the compiler version..

PSR-02: mappings can be packed in a struct

Type	Severity	Location
Gas Optimization	Informational	PaymentSplitter.sol L35, L37

Description:

The mappings on the aforementioned lines have key of type `address` representing a payee. These mappings can be combined into a single mapping having `address` as key type and the value type will be a struct having properties from both of the mappings. This will reduce the lookup gas cost for these mappings

Recommendation:

We advise to replace the aforementioned mappings with a single mapping by utilizing a struct for the value types across all the aforementioned mappings.

```
struct Payee {
   uint256 share;
   mapping(address => uint256) released;
}
```

```
mapping(address => Payee) public payees;
```

Alleviation:

The finding was acknowledged but the team decided to not apply the alleviations.

PSR-03: Inefficient storage access

Type	Severity	Location
Gas Optimization	Informational	PaymentSplitter.sol L106, L109

Description:

The aforementioned lines read storage data `released[_payee][_asset]` twice, which is inefficient as reading from costs around 800 gas.

Recommendation:

We recommend to store the storage data read by `released[_payee][_asset]` in a local variable and then utilize it on the aforementioned to save gas cost associated with extra storage read.

Alleviation:

Alleviations were applied as advised.

PSR-04: Explicitly returning local variable

Type	Severity	Location
Gas Optimization	Informational	PaymentSplitter.sol L99

Description:

The function on the aforementioned line explicitly returns a local variable which increases the overall cost of gas.

Recommendation:

Since named return variables can be declared in the signature of a function, consider refactoring to remove the local variable declaration and explicit return statement in order to reduce the overall cost of gas.

Alleviation:

Alleviations were applied as advised.

PSR-05: Confusing function name

Type	Severity	Location
Coding Style	Informational	PaymentSplitter.sol L99

Description:

The function name `_calculate` on the aforementioned line is confusing which should be replaced with a more comprehensive name reflecting the functionality of the function to increase legibility of the code.

Recommendation:

We advise to rename the function on the aforementioned line which reflects that the function not only calculates the releasing amount but also updates the contract's storage with releasing amount.

Alleviation:

Alleviations were applied by changing the function's name to `_calculateAndUpdateReleasedTokens`.

Appendix

Finding Categories

Gas Optimization

Gas Optimization findings refer to exhibits that do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation exhibits entail findings that relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings are exhibits that detail a fault in the logic of the linked code, such as an incorrect notion on how `block.timestamp` works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a `struct` assignment operation affecting an in-memory `struct` rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of `private` or `delete`.

Coding Style

Coding Style findings usually do not affect the generated byte-code and comment on how to make the codebase more legible and as a result easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a `constructor` assignment imposing different `require` statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as `constant` contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Dead Code

Code that otherwise does not affect the functionality of the codebase and can be safely omitted.