CERTIK

Ignition

Security Assessment

February 24th, 2021

For :
Paid Network

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	Ignition
Description	A typical crowd-sale smart contract.
Platform	Ethereum; Solidity, Yul
Codebase	GitHub Repository
Commits	1. 3877226ab6323ce1cf4d58d0e368407e1e8ad2b1 2. 49f0c3a9c431f723f89ef87de3a5bb59ea9dbf3b

Audit Summary

Delivery Date	February 24th, 2021
Method of Audit	Static Analysis, Manual Review
Consultants Engaged	2
Timeline	February 17th, 2021 - February 24th, 2021

Vulnerability Summary

Total Issues	8
Total Critical	0
Total Major	0
Total Medium	0
Total Minor	2
Total Informational	6

Executive Summary

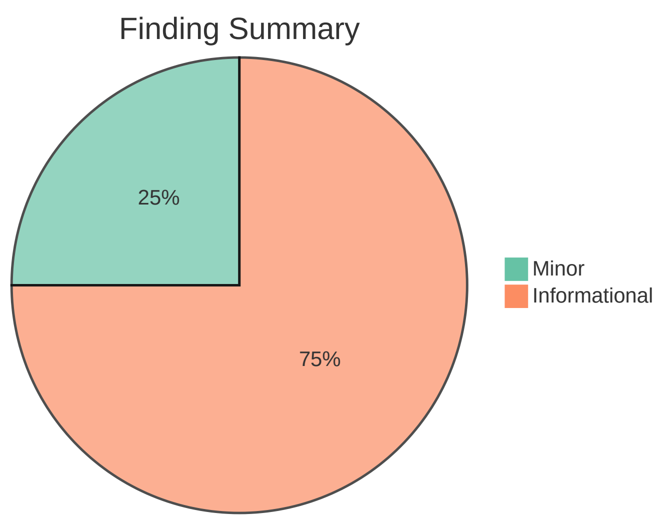
This report represents the results of CertiK's engagement with PAID Network on implementing the Ignition crowd-sale smart contract.

Our findings mainly refer to optimizations and Solidity coding standards; hence the issues identified pose no threat to the contract deployment's safety.

Files In Scope

ID	Contract	Location
IGN	Ignition.sol	contracts/Ignition.sol

Findings



ID	Title	Type	Severity	Resolved
IGN-01	`struct` Optimization	Gas Optimization	Informational	✓
IGN-02	Redundant Variable Initialization	Coding Style	Informational	✓
IGN-03	Inefficient Greater-Than Comparison w/ Zero	Gas Optimization	Informational	✓
IGN-04	Requisite Value of ERC-20 `transferFrom()` / `transfer()` Call	Logical Issue	Minor	✓
IGN-05	Redundant Type Cast	Gas Optimization	Informational	✓
IGN-06	Alternative Assignment	Coding Style	Informational	✓
IGN-07	Redundant State Variable	Data Flow	Informational	✓
IGN-08	Ambiguous Functionality	Volatile Code	Minor	✓

IGN-01: `struct` Optimization

Type	Severity	Location
Gas Optimization	Informational	Ignition.sol L9-L16

Description:

The members of the `Whitelist` struct are not tightly packed.

Recommendation:

We advise to group the `address` and `bool` types together.

Alleviation:

The development team opted to consider our references and strived for a 256-bit packing on the `Whitelist` struct members.

IGN-02: Redundant Variable Initialization

Type	Severity	Location
Coding Style	Informational	Ignition.sol L31, L32, L33

Description:

All variable types within Solidity are initialized to their default "empty" value, which is usually their zeroed out representation. Particularly:

- `uint` / `int`: All `uint` and `int` variable types are initialized at `0`
- `address`: All `address` types are initialized to `address(0)`
- `byte`: All `byte` types are initialized to their `byte(0)` representation
- `bool`: All `bool` types are initialized to `false`
- `ContractType`: All contract types (i.e. for a given `contract ERC20 {}` its contract type is `ERC20`) are initialized to their zeroed out address (i.e. for a given `contract ERC20 {}` its default value is `ERC20(address(0))`)
- `struct`: All `struct` types are initialized with all their members zeroed out according to this table

Recommendation:

We advise that the linked initialization statements are removed from the codebase to increase legibility.

Alleviation:

The development team opted to consider our references and removed the redundant variable initializations.

IGN-03: Inefficient Greater-Than Comparison w/ Zero

Type	Severity	Location
Gas Optimization	Informational	Ignition.sol L118

Description:

The linked greater-than comparisons with zero compare variables that are restrained to the non-negative integer range, meaning that the comparator can be changed to an inequality one which is more gas efficient.

Recommendation:

We advise that the above paradigm is applied to the linked greater-than statements.

Alleviation:

The development team acknowledged this exhibit, but opted to entirely remove the functionality wrapping the linked conditional.

IGN-04: Requisite Value of ERC-20 `transferFrom()` / `transfer()` Call

Type	Severity	Location
Logical Issue	Minor	Ignition.sol L190

Description:

While the ERC-20 implementation does necessitate that the `transferFrom()` / `transfer()` function returns a `bool` variable yielding `true`, many token implementations do not return anything i.e. Tether (USDT) leading to unexpected halts in code execution.

Recommendation:

We advise that the `SafeERC20.sol` library is utilized by OpenZeppelin to ensure that the `transferFrom()` / `transfer()` function is safely invoked in all circumstances.

Alleviation:

The development team opted to consider our references and used the `SafeERC20` library.

IGN-05: Redundant Type Cast

Type	Severity	Location
Gas Optimization	Informational	Ignition.sol L238

Description:

The linked statement redundantly casts the global variable `msg.value` to `uint256`, as it is already of that data type.

Recommendation:

We advise to omit the type cast in the linked statement.

Alleviation:

The development team opted to consider our references and removed the redundant data type cast.

IGN-06: Alternative Assignment

Type	Severity	Location
Coding Style	Informational	Ignition.sol L161

Description:

The linked statement sets the `oneEther` variable equal to `1 ether`.

Recommendation:

We advise to use the global variable `ether` instead, striving for code readability.

Alleviation:

The development team opted to consider our references and set the `oneEther` variable equal to `1 ether`.

IGN-07: Redundant State Variable

Type	Severity	Location
Data Flow	Informational	Ignition.sol L28

Description:

The `whitelistAddresses` array is introduced to store the whitelisted addresses of the system, yet it is not used direct by the contract.

Recommendation:

We advise to index the events off-chain, instead of storing extra data on-chain.

Alleviation:

The development team opted to consider our references, removed the `whitelistAddresses` array and decided to handle the events off-chain instead.

IGN-08: Ambiguous Functionality

Type	Severity	Location
Volatile Code	Minor	Ignition.sol L236-L254

Description:

A whitelisted user can buy tokens even after the end of the sale, as the linked function only checks against the starting sale time.

Recommendation:

We advise to either revise the linked function or add descriptive documentation for the edge case.

Alleviation:

The development team opted to consider our references and added a `require` statement checking that the sale period is finished.

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.