Kira Core

Security Assessment

November 18th, 2020

For :
Kira Core

By :
Alex Papageorgiou @ CertiK
alex.papageorgiou@certik.org

Angelos Apostolidis @ CertiK
angelos.apostolidis@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	Kira Core
Description	This audit is based on the staking and auction contracts of the Kira Liquidity program
Platform	Ethereum; Solidity, Yul
Codebase	GitHub Repository
Commits	pre-audit: 7ed63aed26679d855abd24230d63d381118ffd29 post-audit: 11494f85bc25f5a8a3c71536bbf004fe05a45bdf

Audit Summary

Delivery Date	November 18th, 2020
Method of Audit	Static Analysis, Manual Review
Consultants Engaged	2
Timeline	October 23rd, 2020 - November 18th, 2020

Vulnerability Summary

Total Issues	12
Total Critical	0
Total Major	0
Total Medium	0
Total Minor	0
Total Informational	12

Executive Summary

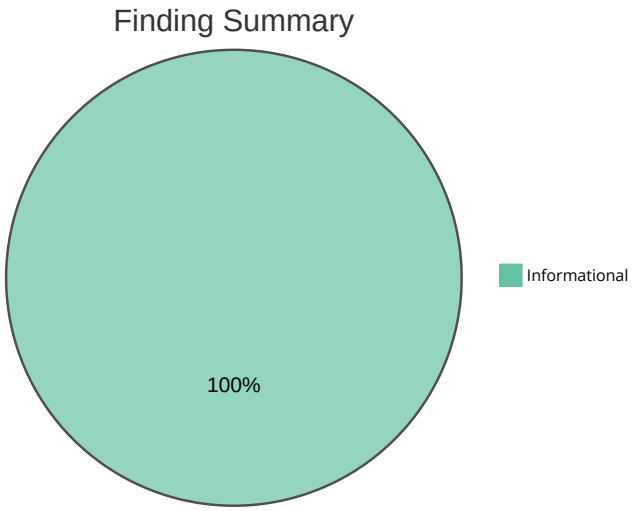
This report represents the results of CertiK's engagement with Kira Core on their implementation of the staking and auction contracts of the Kira Liquidity program .

Our findings mainly refer to optimizations and Solidity coding standards. Hence, the issues identified pose no threat to the safety of the contract deployment.

Files In Scope

ID	Contract	Location
KSG	KiraStaking.sol	LIP_2/contracts/KiraStaking.sol
OWN	Owned.sol	LIP_2/contracts/Owned.sol
PAU	Pausable.sol	LIP_2/contracts/Pausable.sol
KAN	KiraAuction.sol	LIP_3/contracts/KiraAuction.sol

Findings



ID	Title	Type	Severity	Resolved
KSG-01	Redundant Variable Initialization	Coding Style	Informational	✓
KSG-02	User-Defined Getters	Gas Optimization	Informational	✓
KSG-03	`private` over `public` mapping	Gas Optimization	Informational	✓
KSG-04	State Variables Layout	Gas Optimization	Informational	✓
KAN-01	Redundant Variable Initialization	Coding Style	Informational	✓
KAN-02	State Variables Layout	Gas Optimization	Informational	✓
KAN-03	User-Defined Getters	Gas Optimization	Informational	✓
KAN-04	Function Optimization	Gas Optimization	Informational	✓
KAN-05	Conditional Optimization	Gas Optimization	Informational	✓
KAN-06	Function Optimization	Gas Optimization	Informational	✓
KAN-07	Function Extension	Coding Style	Informational	ⓘ
KAN-08	Function Optimization	Gas Optimization	Informational	✓

KSG-01: Redundant Variable Initialization

Type	Severity	Location
Coding Style	Informational	KiraStaking.sol L20-L22, L25

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 initializations.

KSG-02: User-Defined Getters

Type	Severity	Location
Gas Optimization	Informational	KiraStaking.sol L30, L46-L48

Description:

The linked variables contain user-defined getter functions that are equivalent to their name barring for an underscore (`_`) prefix / suffix.

Recommendation:

We advise that the linked variables are instead declared as `public` and that they are renamed to their respective getter's name as compiler-generated getter functions are less prone to error and much more maintainable than manually written ones.

Alleviation:

The development team opted to consider our references, changed the linked state variables to `public` and removed the manual getter function(s).

KSG-03: `private` over `public` mapping

Type	Severity	Location
Gas Optimization	Informational	KiraStaking.sol L27, L28

Description:

The linked `mappings` have the `public` visibility specifier, but could be restricted to `private` to save gas.

Recommendation:

We advise that the linked variables use the `private` visibility specifier.

Alleviation:

The development team opted to consider our references and changed the linked `mapping`s to `private`, leading to gas save.

KSG-04: State Variables Layout

Type	Severity	Location
Gas Optimization	Informational	KiraStaking.sol L18-L31

Description:

The layout of the state variables should be as tightly packed as possible, in order to save gas.

Recommendation:

We advise the team to change the state variable layout to:

```
uint256 public periodFinish = 0;
uint256 public rewardRate = 0;
uint256 public rewardsDuration = 0;
uint256 public lastUpdateTime;
uint256 public rewardPerTokenStored;
uint256 public lastBalance = 0;
uint256 private _totalSupply;

IERC20 public rewardsToken;
IERC20 public stakingToken;

mapping(address => uint256) public userRewardPerTokenPaid;
mapping(address => uint256) public rewards;
mapping(address => uint256) private _balances;
```

Alleviation:

The development team opted to consider our references and changed to an optimal state variable layout.

KAN-01: Redundant Variable Initialization

Type	Severity	Location
Coding Style	Informational	KiraAuction.sol L32, L38-L40, L55-L58, L84, L179, L313

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 initializations.

KAN-02: State Variables Layout

Type	Severity	Location
Gas Optimization	Informational	KiraAuction.sol L30-L58

Description:

The layout of the state variables should be as tightly packed as possible, in order to save gas.

Recommendation:

We advise the team to change the state variable layout to:

```
struct UserInfo {
    bool whitelisted;
    uint256 claimed_wei;
    uint256 last_deposit_time;
    bool claimed;
    bool distributed;
}

uint256 public startTime = 0;
uint256 private P1;
uint256 private P2;
uint256 private P3;
uint256 private T1;
uint256 private T2;
uint256 private MIN_WEI = 0 ether;
uint256 private MAX_WEI = 0 ether;
uint256 private INTERVAL_LIMIT = 0;
uint256 private totalWeiAmount = 0;
uint256 private latestPrice = 0;

address payable public wallet;
bool public isFinished = false;
ERC20 private kiraToken;
mapping(address => UserInfo) private customers;
address[] private arrayAddress;
```

Alleviation:

The development team opted to consider our references and changed to an optimal state variable layout.

KAN-03: User-Defined Getters

Type	Severity	Location
Gas Optimization	Informational	KiraAuction.sol L56, L157-L159

Description:

The linked variables contain user-defined getter functions that are equivalent to their name barring for an underscore (`_`) prefix / suffix.

Recommendation:

We advise that the linked variables are instead declared as `public` and that they are renamed to their respective getter's name as compiler-generated getter functions are less prone to error and much more maintainable than manually written ones.

Alleviation:

The development team opted to consider our references, changed the linked state variables to `public` and removed the manual getter function(s).

KAN-04: Function Optimization

Type	Severity	Location
Gas Optimization	Informational	KiraAuction.sol L119-L137

Description:

The linked function executes redundant `mapping` look-ups, hence increacing the gas amount.

Recommendation:

We advise the team to change the linked function to:

```
function getCustomerInfo(address addr) external view
  returns (
    bool,
    uint256,
    uint256,
    bool,
    bool
  ){
    UserInfo storage customer = customers[addr];
    return (
      customer.whitelisted,
      customer.claimed_wei,
      customer.last_deposit_time,
      customer.claimed,
      customer.distributed
    );
}
```

Alleviation:

The development team opted to consider our references, saved the `UserInfo` to `storage` and removed the redundant `mapping` look-ups.

KAN-05: Conditional Optimization

Type	Severity	Location
Gas Optimization	Informational	KiraAuction.sol L244-L245

Description:

The linked require statements can be further optimized.

Recommendation:

We advise the team to follow the pattern used in L243 when checking the value of the `_t1` and `_t2` parameters.

```
require(_t2 > _t1 && _t1 > 0, "Error Message");
```

Alleviation:

The development team opted to consider our references and optimized the conditional by removing a redundant `require` statement.

KAN-06: Function Optimization

Type	Severity	Location
Gas Optimization	Informational	KiraAuction.sol L312-L341

Description:

The linked function can be further optimized.

Recommendation:

We advise the team to remove redundant `mapping` look-ups and `memory` assignments.

```
function distribute() external onlyOwner onlyAfterAuction {
    uint256 totalDistributed = 0;
    uint256 exp = 10**uint256(kiraToken.decimals());
    uint256 numberOfContributors = arrayAddress.length;

    for (uint256 i = 0; i < numberOfContributors; i++) {
        address addr = arrayAddress[i];
        UserInfo storage customer = customers[addr];

        if (customer.claimed_wei > 0 && !customer.claimed &&
!customer.distributed) {
            uint256 tokensToSend =
customer.claimed_wei.mul(exp).div(latestPrice);
            uint256 currentBalance = kiraToken.balanceOf(address(this));

            customer.distributed = true;

            if (currentBalance < tokensToSend) {
                tokensToSend = currentBalance;
            }

            if (tokensToSend > 0) {
                totalDistributed = totalDistributed.add(tokensToSend);
                kiraToken.transfer(addr, tokensToSend);
            }
        }
    }

    require(totalDistributed > 0, 'KiraAuction: nothing to distribute. already
claimed or distributed all!');

    emit DistributeTokens(totalDistributed);
}
```

Alleviation:

The development team opted to consider our references, saved the `UserInfo` to `storage` and removed the redundant `mapping` look-ups.

KAN-07: Function Extension

Type	Severity	Location
Coding Style	Informational	KiraAuction.sol L312-L341

Description:

The `distribute()` function iterates over an array of addresses and distributes tokens, if available, serially.

Recommendation:

We advise the team to return the index of the array if the balance of the Kira Token contract drops lower than the amount-to-be-distributed, indicating how many of the customers were served during the procedure.

Alleviation:

The Kira Core development team has acknowledged this exhibit but decided to not apply its remediation in the current version of the codebase due to time constraints.

KAN-08: Function Optimization

Type	Severity	Location
Gas Optimization	Informational	KiraAuction.sol L343-L356

Description:

The linked function can be further optimized.

Recommendation:

We advise the team to remove redundant `mapping` look-ups and `memory` assignments.

```
function claimTokens() external onlyAfterAuction {
    UserInfo storage customer = customers[msg.sender];
    require(!customer.claimed, 'KiraAuction: you claimed already.');
    require(!customer.distributed, 'KiraAuction: we already sent to your
wallet.');
    require(customer.whitelisted && (customer.claimed_wei > 0), 'KiraAuction: you
did not contribute.');

    customer.claimed = true;

    uint256 exp = 10**uint256(kiraToken.decimals());
    uint256 amountToClaim = customer.claimed_wei.mul(exp).div(latestPrice);
    kiraToken.transfer(msg.sender, amountToClaim);

    emit ClaimedTokens(msg.sender, amountToClaim);
}
```

Alleviation:

The development team opted to consider our references, saved the `UserInfo` to `storage`, removed the redundant `mapping` look-ups and optimized the `require` statements.

Appendix

Icons explanation

 : Issue resolved

 : Issue not resolved / Acknowledged. The team will be fixing the issues in the own timeframe.

 : Issue partially resolved. Not all instances of an issue was resolved.

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.