Kingswap ERC20 Smart Contract

AUDIT REPORT

Prepared By :

Celticlab Private Limited

Corporate Office:

H. No : 45

Awas Vikas Colony

Basti

U.P

Pin : 272001

India

CIN – U72900UP2017PTC097710

Email ID - himanshu@celticlab.com

Introduction

This is a technical audit for Kingswap (V1) smart contracts. This document outlines our methodology, limitations and results for our security audit. The solidity files covered are :

1. KingToken.sol
2. Archbishop.sol
3. StakeHolderFund.sol

All activities were conducted in a way which aimed to simulate the mindset of a malicious actor engaging in a targeted attack towards the above mentioned smart contracts with the expected goals of:

- Identifying ways to manipulate the state modifying logic of the smart contract.
- Shedding light on bad coding practices.
- Determining the impact of external malicious behaviour.

Due to the nature of smart contracts, the importance of security cannot be overstated. Once a smart contract goes live, it will be very challenging to correct any mistakes. This furthermore enhances the importance of auditing code, skipping this step, raises a great risk of potential value loss.

Synopsis

In regards to modularity and simplicity, it is a good practice to keep contracts and functions small, and in the concerned smart contracts the such practices are followed. It is recommended to prefer clarity over performance whenever possible and use existing code from contracts such as the ones provided by OpenZeppelin which is thoroughly audited and tested, anywhere possible. OpenZeppelin contracts are used in the implementation. Overall, the code demonstrates high code quality standards adopted and effective use of concept and modularity. The contract development team demonstrated high technical capabilities, both in the design of the architecture and in the implementation.

Code Analysis

Besides, the results of the automated analysis, manual verification was also taken into account. The complete contract was manually analyzed, every logic was checked and compared with the comments made in the contract. The manual analysis of code confirms that the Contract does not contain any serious susceptibility. No divergence was found between the logic in Smart Contract and the informative smart contract comments.

Scope

This audit is into the technical and security aspects of the Kingswap smart contracts. The key aim of this audit is to ensure that transactions taking place in these contracts are not by far attacked or misused by any third party and they occur in the right intention of the transaction initiator. The next aim of this audit is to ensure the coded algorithms work as expected. The audit of Smart Contract also checks the implementation of token mechanism i.e. The KingToken contract must follow the ERC20 Standard.

This audit is purely technical and is not investment advice. The scope of the audit is limited to the below source codes:

1. KingToken.sol

```
pragma solidity 0.6.12;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/GSN/Context.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Address.sol";

// KingToken with Governance.
contract KingToken is Context, IERC20, Ownable {
    using SafeMath for uint256;
    using Address for address;

    mapping (address => uint256) private _balances;

    mapping (address => mapping (address => uint256)) private _allowances;

    uint256 private _totalSupply;

    string private _name = "KingToken";
    string private _symbol = "KING";
    uint8 private _decimals = 18;

    /**
     * @dev Returns the name of the token.
     */
    function name() public view returns (string memory) {
        return _name;
    }

    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() public view returns (string memory) {
        return _symbol;
    }

    /**
     * @dev Returns the number of decimals used to get its user representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens should
     * be displayed to a user as `5,05` (`505 / 10 ** 2`).
     *
     * Tokens usually opt for a value of 18, imitating the relationship between
     * Ether and Wei. This is the value {ERC20} uses, unless {_setupDecimals} is
     * called.
     *
     * NOTE: This information is only used for _display_ purposes: it in
     * no way affects any of the arithmetic of the contract, including
     * {IERC20-balanceOf} and {IERC20-transfer}.
     */
    function decimals() public view returns (uint8) {
        return _decimals;
    }
```

```
/**
     * @dev See {IERC20-totalSupply}.
     */
    function totalSupply() public view override returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev See {IERC20-balanceOf}.
     */
    function balanceOf(address account) public view override returns (uint256) {
        return _balances[account];
    }

    /**
     * @dev See {IERC20-transfer}.
     *
     * Requirements:
     *
     * - `recipient` cannot be the zero address.
     * - the caller must have a balance of at least `amount`.
     */
    function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
    }

    /**
     * @dev See {IERC20-allowance}.
     */
    function allowance(address owner, address spender) public view virtual override returns (uint256) {
        return _allowances[owner][spender];
    }

    /**
     * @dev See {IERC20-approve}.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public virtual override returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
    }
```

```
/**
     * @dev See {IERC20-transferFrom}.
     *
     * Emits an {Approval} event indicating the updated allowance. This is not
     * required by the EIP. See the note at the beginning of {ERC20};
     *
     * Requirements:
     * - `sender` and `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     * - the caller must have allowance for ``sender``'s tokens of at least
     * `amount`.
     */
    function transferFrom(address sender, address recipient, uint256 amount) public virtual override
returns (bool) {
        _transfer(sender, recipient, amount);
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer
amount exceeds allowance"));
        return true;
    }

    /**
     * @dev Atomically increases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
        return true;
    }

    /**
     * @dev Atomically decreases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
{
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "ERC20:
decreased allowance below zero"));
        return true;
    }
```

```
/**
     * @dev Moves tokens `amount` from `sender` to `recipient`.
     *
     * This is internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */
    function _transfer(address sender, address recipient, uint256 amount) internal virtual {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer(sender, recipient, amount);

        _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);

        _moveDelegates(_delegates[sender], _delegates[recipient], amount);
    }

    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * Requirements
     *
     * - `to` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");

        _beforeTokenTransfer(address(0), account, amount);

        _totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(address(0), account, amount);
    }
```

```
/**
     * @dev Destroys `amount` tokens from `account`, reducing the
     * total supply.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * Requirements
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");

        _beforeTokenTransfer(account, address(0), amount);

        _balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds balance");
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), amount);
    }

    /**
     * @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
     *
     * This is internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(address owner, address spender, uint256 amount) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    /**
     * @dev Sets {decimals} to a value other than the default one of 18.
     *
     * WARNING: This function should only be called from the constructor. Most
     * applications that interact with token contracts will not expect
     * {decimals} to ever change, and may work incorrectly if it does.
     */
    function _setupDecimals(uint8 decimals_) internal {
        _decimals = decimals_;
    }
```

```
/**
     * @dev Hook that is called before any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * will be to transferred to `to`.
     * - when `from` is zero, `amount` tokens will be minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
     * - `from` and `to` are never both zero.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
     */
    function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual { }

    /// @notice Creates `_amount` token to `_to`. Must only be called by the owner (Archbishop).
    function mint(address _to, uint256 _amount) public onlyOwner {
        _mint(_to, _amount);
        _moveDelegates(address(0), _delegates[_to], _amount);
    }

    // Copied and modified from YAM code:
    // https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernanceStorage.sol
    // https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernance.sol
    // Which is copied and modified from COMPOUND:
    // https://github.com/compound-finance/compound-protocol/blob/master/contracts/Governance/Comp.sol

    /// @notice A record of each accounts delegate
    mapping (address => address) internal _delegates;

    /// @notice A checkpoint for marking number of votes from a given block
    struct Checkpoint {
        uint32 fromBlock;
        uint256 votes;
    }

    /// @notice A record of votes checkpoints for each account, by index
    mapping (address => mapping (uint32 => Checkpoint)) public checkpoints;

    /// @notice The number of checkpoints for each account
    mapping (address => uint32) public numCheckpoints;

    /// @notice The EIP-712 typehash for the contract's domain
    bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256
chainId,address verifyingContract)");
```

```
/// @notice The EIP-712 typehash for the delegation struct used by the contract
    bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256
nonce,uint256 expiry)");

    /// @notice A record of states for signing / validating signatures
    mapping (address => uint) public nonces;

    /// @notice An event thats emitted when an account changes its delegate
    event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed
toDelegate);

    /// @notice An event thats emitted when a delegate account's vote balance changes
    event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

    /**
     * @notice Delegate votes from `msg.sender` to `delegatee`
     * @param delegator The address to get delegatee for
     */
    function delegates(address delegator)
        external
        view
        returns (address)
    {
        return _delegates[delegator];
    }

   /**
    * @notice Delegate votes from `msg.sender` to `delegatee`
    * @param delegatee The address to delegate votes to
    */
    function delegate(address delegatee) external {
        return _delegate(msg.sender, delegatee);
    }

    /**
     * @notice Delegates votes from signatory to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(
        address delegatee,
        uint nonce,
        uint expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
        external
    {
        bytes32 domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name())),
                getChainId(),
                address(this)
            )
        );
```

```
bytes32 structHash = keccak256(
            abi.encode(
                DELEGATION_TYPEHASH,
                delegatee,
                nonce,
                expiry
            )
        );

        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator,
                structHash
            )
        );

        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "KING::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "KING::delegateBySig: invalid nonce");
        require(now <= expiry, "KING::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }

    /**
     * @notice Gets the current votes balance for `account`
     * @param account The address to get votes balance
     * @return The number of current votes for `account`
     */
    function getCurrentVotes(address account)
        external
        view
        returns (uint256)
    {
        uint32 nCheckpoints = numCheckpoints[account];
        return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
    }
```

```
/**
     * @notice Determine the prior number of votes for an account as of a block number
     * @dev Block number must be a finalized block or else this function will revert to prevent
misinformation.
     * @param account The address of the account to check
     * @param blockNumber The block number to get the vote balance at
     * @return The number of votes the account had as of the given block
     */
    function getPriorVotes(address account, uint blockNumber)
        external
        view
        returns (uint256)
    {
        require(blockNumber < block.number, "KING::getPriorVotes: not yet determined");

        uint32 nCheckpoints = numCheckpoints[account];
        if (nCheckpoints == 0) {
            return 0;
        }

        // First check most recent balance
        if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
            return checkpoints[account][nCheckpoints - 1].votes;
        }

        // Next check implicit zero balance
        if (checkpoints[account][0].fromBlock > blockNumber) {
            return 0;
        }

        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
            Checkpoint memory cp = checkpoints[account][center];
            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return checkpoints[account][lower].votes;
    }

    function _delegate(address delegator, address delegatee)
        internal
    {
        address currentDelegate = _delegates[delegator];
        uint256 delegatorBalance = balanceOf(delegator); // balance of underlying KINGs (not scaled);
        _delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }
```

```
function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                // decrease old representative
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }

    function _writeCheckpoint(
        address delegatee,
        uint32 nCheckpoints,
        uint256 oldVotes,
        uint256 newVotes
    )
        internal
    {
        uint32 blockNumber = safe32(block.number, "KING::_writeCheckpoint: block number exceeds 32
bits");

        if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
            checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
        } else {
            checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
            numCheckpoints[delegatee] = nCheckpoints + 1;
        }

        emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
    }

    function safe32(uint n, string memory errorMessage) internal pure returns (uint32) {
        require(n < 2**32, errorMessage);
        return uint32(n);
    }

    function getChainId() internal pure returns (uint) {
        uint256 chainId;
        assembly { chainId := chainid() }
        return chainId;
    }
}
```

2. Archbishop.sol

```solidity
pragma solidity 0.6.12;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/utils/EnumerableSet.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "./KingToken.sol";
import "./lib/UQ112x112.sol";

interface IMigratorChef {
    // Perform LP token migration from legacy UniswapV2 to KingSwap.
    // Take the current LP token address and return the new LP token address.
    // Migrator should have full access to the caller's LP token.
    // Return the new LP token address.
    //
    // XXX Migrator must have allowance access to UniswapV2 LP tokens.
    // KingSwap must mint EXACTLY the same amount of KingSwap LP tokens or
    // else something bad will happen. Traditional UniswapV2 does not
    // do that so be careful!
    function migrate(IERC20 token) external returns (IERC20);
}

// Archbishop will crown the King and he is a fair guy.
//
// Note that it's ownable and the owner wields tremendous power. The ownership
// will be transferred to a governance smart contract once KING is sufficiently
// distributed and the community can show to govern itself.
//
// Have fun reading it. Hopefully it's bug-free. God bless.
contract Archbishop is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;
    using UQ112x112 for uint112;

    // Info of each user.
    struct UserInfo {
        uint256 amount; // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt. See explanation below.
        //
        // We do some fancy math here. Basically, any point in time, the amount of KINGs
        // entitled to a user but is pending to be distributed is:
        //
        //   pending reward = (user.amount * pool.accKingPerShare) - user.rewardDebt
        //
        // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
        //   1. The pool's `accKingPerShare` (and `lastRewardBlock`) gets updated.
        //   2. User receives the pending reward sent to his/her address.
        //   3. User's `amount` gets updated.
        //   4. User's `rewardDebt` gets updated.
    }
```

```
// Info of each pool.
    struct PoolInfo {
        IERC20 lpToken; // Address of LP token contract.
        uint256 allocPoint; // How many allocation points assigned to this pool. KINGs to distribute
per block.
        uint256 lastRewardBlock; // Last block number that KINGs distribution occurs.
        uint256 accKingPerShare; // Accumulated KINGs per share, times 1e12. See below.
    }

    // The KING TOKEN!
    KingToken public king;

    // Stakeholder Address
    address public stakeholderaddress;
    // Block number when bonus KING period ends.
    uint256 public bonusEndBlock;
    // KING tokens created per block.
    uint256 public kingPerBlock;
    // Bonus muliplier for early king makers.
    uint256 public constant BONUS_MULTIPLIER = 10;
    // Bonus block num,about 15 days.
    uint256 public constant BONUS_BLOCKNUM = 64000;
    // The migrator contract. It has a lot of power. Can only be set through governance (owner).
    IMigratorChef public migrator;

    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;
    // Record whether the pair has been added.
    mapping(address => uint256) public lpTokenPID;
    // Total allocation poitns. Must be the sum of all allocation points in all pools.
    uint256 public totalAllocPoint = 0;
    // The block number when KING mining starts.
    uint256 public startBlock;

    event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
    event EmergencyWithdraw(
        address indexed user,
        uint256 indexed pid,
        uint256 amount
    );

    constructor(
        KingToken _king,
        address _stakeholderaddress,
        uint256 _kingPerBlock,
        uint256 _startBlock
    ) public {
        king = _king;
        stakeholderaddress = _stakeholderaddress;
        kingPerBlock = _kingPerBlock;
        startBlock = _startBlock;
        bonusEndBlock = startBlock.add(BONUS_BLOCKNUM);
    }
```

```
function poolLength() external view returns (uint256) {
        return poolInfo.length;
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        require(lpTokenPID[address(_lpToken)] == 0, "Archbishop:duplicate add.");
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accKingPerShare: 0
            })
        );
        lpTokenPID[address(_lpToken)] = poolInfo.length;
    }

    // Update the given pool's KING allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    // Set the migrator contract. Can only be called by the owner.
    function setMigrator(IMigratorChef _migrator) public onlyOwner {
        migrator = _migrator;
    }

    // Migrate lp token to another lp contract. Can be called by anyone. We trust that migrator
contract is good.
    function migrate(uint256 _pid) public {
        require(address(migrator) != address(0), "migrate: no migrator");
        PoolInfo storage pool = poolInfo[_pid];
        IERC20 lpToken = pool.lpToken;
        uint256 bal = lpToken.balanceOf(address(this));
        lpToken.safeApprove(address(migrator), bal);
        IERC20 newLpToken = migrator.migrate(lpToken);
        require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
        pool.lpToken = newLpToken;
    }
```

```
// Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to) public view returns (uint256) {
        if (_to <= bonusEndBlock) {
            return _to.sub(_from).mul(BONUS_MULTIPLIER);
        } else if (_from >= bonusEndBlock) {
            return _to.sub(_from);
        } else {
            return bonusEndBlock.sub(_from).mul(BONUS_MULTIPLIER).add(
                _to.sub(bonusEndBlock)
            );
        }
    }

    // View function to see pending KINGs on frontend.
    function pendingKing(uint256 _pid, address _user) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accKingPerShare = pool.accKingPerShare;
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (block.number > pool.lastRewardBlock && lpSupply != 0) {
            uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
            uint256 kingReward =
multiplier.mul(kingPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
            uint256 kingReward2nd = kingReward.mul(9).div(25);

            kingReward = kingReward.sub(kingReward2nd);
            uint256 balance = 0;

            // Balance in Archbishop less than 10 due to corrections will be placed back into pool.
            if(king.balanceOf(address(this))  <= 10){
                balance = king.balanceOf(address(this));
            }

            accKingPerShare = accKingPerShare.add(kingReward.add(balance).mul(1e12).div(lpSupply));

        }
        return user.amount.mul(accKingPerShare).div(1e12).sub(user.rewardDebt);
    }

    // Update reward vairables for all pools. Be careful of gas spending!
    function massUpdatePools() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            updatePool(pid);
        }
    }
```

```
// Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        if (multiplier == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }


        // King Rewards meant for Community
        uint256 kingReward = multiplier.mul(kingPerBlock).mul(pool.allocPoint).div(totalAllocPoint);

        // King Rewards meant for Stake Holders

        uint256 kingReward2nd = kingReward.mul(9).div(25);

        kingReward = kingReward.sub(kingReward2nd);

        uint256 balance = 0;

        // Balance in Archbishop less than 10 due to corrections will be placed back into pool.
        if(king.balanceOf(address(this))  <= 10){
            balance = king.balanceOf(address(this));
        }

        king.mint(stakeholderaddress, kingReward2nd);
        king.mint(address(this), kingReward);

        pool.accKingPerShare =
pool.accKingPerShare.add(kingReward.add(balance).mul(1e12).div(lpSupply));
        pool.lastRewardBlock = block.number;

    }

    // Deposit LP tokens to Archbishop for KING allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        uint256 pending = user.amount.mul(pool.accKingPerShare).div(1e12).sub(user.rewardDebt);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accKingPerShare).div(1e12);
        if (pending > 0) safeKingTransfer(msg.sender, pending);
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        emit Deposit(msg.sender, _pid, _amount);
    }
```

```
// Withdraw LP tokens from Archbishop.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pending = user.amount.mul(pool.accKingPerShare).div(1e12).sub(user.rewardDebt);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accKingPerShare).div(1e12);
        safeKingTransfer(msg.sender, pending);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _pid, _amount);
    }

    // Withdraw without caring about rewards. EMERGENCY ONLY. // what is the difference?
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount > 0, "emergencyWithdraw: not good");
        uint256 _amount = user.amount;
        user.amount = 0;
        user.rewardDebt = 0;
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit EmergencyWithdraw(msg.sender, _pid, _amount);
    }

    // Safe king transfer function, just in case if rounding error causes pool to not have enough
KINGs.
    function safeKingTransfer(address _to, uint256 _amount) internal {
        uint256 kingBal = king.balanceOf(address(this));
        if (_amount > kingBal) {
            king.transfer(_to, kingBal);
        } else {
            king.transfer(_to, _amount);
        }
    }



    // Update dev address by the previous dev.
    function stakeholder(address _stakeholderaddress) public {
        require(msg.sender == stakeholderaddress, "stakeholder: wut?");
        stakeholderaddress = _stakeholderaddress;
    }


}
```

3. StakeHolderFund.sol

```
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "./KingToken.sol";

contract StakeHolderFund is Ownable {
    using SafeMath for uint;

    // the king token
    KingToken public king;

    //Early Liquidity Provider
    struct ELPInfo {
        address walletAddress;
        uint index;
        uint256 allocPoint; // How many allocation points assigned to this pool. KINGs to distribute
per block.
        bool statusActive;
    }

    struct UserInfo{
        address walletAddress;
        // Assume it as ID rather than Index, will be index-1 in the Array *important*
        uint index;
        // Inactive would mean the wallet will not recieve token
        bool statusActive;
    }

    // Advisor Address.
    // Assume the uint as the Index, will be identical to index in the Array
    mapping(address => uint) public advAddress;
    UserInfo[] public advArray;

    // EarlyLP Address.
    mapping(address => uint) public elpAddress;
    ELPInfo[] public elpArray;

    // eTeam Address.
    mapping(address => uint) public eteamAddress;
    UserInfo[] public eteamArray;

    // Company Address.
    address public companyaddr;

    // StakeholderFund Timelock address
    address public stakeHolderFundTimelock;
```

```
//Default Constructor
    constructor(KingToken _king,
     ELPInfo[]  memory _elpaddr,
     UserInfo[] memory _advaddr,
     UserInfo[] memory _eteamaddr,
     address _companyaddr,
     address _stakeHolderFundTimelock)
    public{
        king = _king;
        //Initialize LPinfo address
        updateELPInfo(_elpaddr,elpArray);
        updateMappingELPInfo(elpAddress,_elpaddr);

        //Initialize userinfo address
        updateUserInfo(_advaddr,advArray);
        updateMappingUserInfo(advAddress,_advaddr);
        updateUserInfo(_eteamaddr,eteamArray);
        updateMappingUserInfo(eteamAddress,_eteamaddr);

        stakeHolderFundTimelock = _stakeHolderFundTimelock;
        companyaddr = _companyaddr;
    }

    //event NewRequest(uint);
    event fundRequest(uint);
    //Load the mapping for first few ELP
    function updateELPInfo(ELPInfo[] memory init,ELPInfo[] storage mainArray) internal {
        uint256 length = init.length;
        for (uint i = 0 ; i < length; i++){
            address walletAddress = init[i].walletAddress;
            uint index = init[i].index;
            uint point = init[i].allocPoint;
            bool status = init[i].statusActive;
            ELPInfo memory e = ELPInfo({
                walletAddress:walletAddress,
                index: index,
                allocPoint: point,
                statusActive: status});

            //store in array checker address. Index will be -1 to represent the position.
            mainArray.push(e);
            //emit NewRequest(elpArray.length);
        }
        //emit NewRequest(elpArray.length);
    }
    function updateMappingELPInfo(mapping(address => uint) storage map,ELPInfo[] memory init) internal{
        // Add all the users into the address
        for (uint i = 0 ; i < init.length; i++){
            // Retrieve wallet address
            address walletAddress = init[i].walletAddress;
            //store in map.
            uint index = i + 1;
            map[walletAddress] = index;
        }
    }
```

```
function updateUserInfo(UserInfo[] memory init,UserInfo[] storage mainArray) internal {
        uint256 length = init.length;
        for (uint i = 0 ; i < length; i++){
            address walletAddress = init[i].walletAddress;
            uint index = init[i].index;
            bool statusActive = init[i].statusActive;
            UserInfo memory e = UserInfo({
                walletAddress: walletAddress,
                index: index,
                statusActive: statusActive});

            //store in array checker address. Index will be -1 to represent the position.
            mainArray.push(e);
        }
    }

    //Initialize the mapping
    function updateMappingUserInfo(mapping(address => uint) storage map,UserInfo[] memory init)
internal{
        // Add all the users into the address
        for (uint i = 0 ; i < init.length; i++){
            // Retrieve wallet address
            address walletAddress = init[i].walletAddress;
            //store in map.
            uint index = i + 1;
            map[walletAddress] = index;
        }
    }


    function getTotalAdv() onlyOwner external view returns(uint){
        return advArray.length;
    }

    function getTotalEteam() onlyOwner external view returns(uint){
        return eteamArray.length;
    }

    function getTotalELP()  external view returns(uint){
        return elpArray.length;
    }

    //calculate total allocation point
    function ELPAllocPoint() internal returns(uint) {
        uint totalAllocPoint = 0;
        for(uint i = 0 ;i < elpArray.length; i++){
            //calculate only for wallet with active status.
            if(elpArray[i].statusActive == true){
                totalAllocPoint = totalAllocPoint + elpArray[i].allocPoint;
            }
        }
        return totalAllocPoint;
    }
```

```
//Anyone can call it but it will be split to only the people in this address.
    function withdraw() public {
        uint _amount = king.balanceOf(address(this));
        require(_amount > 0, "zero king amount");
        uint amountReal = _amount;
        uint totalAllocPoint = ELPAllocPoint();
        uint balance = amountReal;

        //Transfer this amount to this contract address
        uint shFundTimeLock = amountReal.mul(1170).div(3600);
        king.transfer(stakeHolderFundTimelock, shFundTimeLock);

        balance = balance.sub(shFundTimeLock);

        // This contract is suppose to get 3600
        // elp supposed to get 1500 out of 3600
        for(uint i = 0 ; i < elpArray.length; i ++){

            //only for active wallet address
            if(elpArray[i].statusActive == true){
                //Actual indiividual ELP amount : Real Amount * AllocationPoint Per ELP/Total
Allocation * ELP Share/Total Share
                uint fund =
amountReal.mul(elpArray[i].allocPoint).mul(1500).div(totalAllocPoint.mul(3600));
                balance = balance.sub(fund);
                king.transfer(elpArray[i].walletAddress, fund);
            }
        }

        // Advisor suppose to get 100 out of 3600 : Real Amount * Advisor Share/Totalshare
        for(uint i = 0 ; i < advArray.length; i ++){

            //only for active advisor wallet address
            if(advArray[i].statusActive == true){
                //based total advisor in team divide equally
                uint advFund = amountReal.mul(100).div((advArray.length).mul(3600));
                balance = balance.sub(advFund);
                king.transfer(advArray[i].walletAddress, advFund);
            }
        }
```

```
// Eteam suppose to get 250 out of 3600
      for(uint i = 0 ; i < eteamArray.length; i ++){

          //only for active eteam wallet address
          if(eteamArray[i].statusActive == true){

              //based total team members in team divide equally
              uint eteamFund = amountReal.mul(250).div((eteamArray.length).mul(3600));
               emit fundRequest(eteamFund);
              balance = balance.sub(eteamFund);
              king.transfer(eteamArray[i].walletAddress, eteamFund);
          }
      }
      //emit balanceRequest(balance);

      //Company suppose to get 580 out 3600
      king.transfer(companyaddr,balance);
  }
  //event balanceRequest(uint);
  function addAdvAddress(address _advAddress) public onlyOwner{
      if(advAddress[_advAddress]==0){
          uint index = advArray.length;
          if(advArray.length == 0){
              index = 1;
          }
          //Index need to plus as it takes the length
          advAddress[_advAddress] = index+1;
          advArray.push(UserInfo({
              walletAddress: _advAddress,
              index: index,
              statusActive: true}));
      }
  }

  function addelpAddress(address _elpAddress,uint256 _point) public onlyOwner{
      if(elpAddress[_elpAddress]==0){
          uint index = elpArray.length;
          if(elpArray.length == 0){
              index = 1;
          }
          elpAddress[_elpAddress] = index-1;
          elpArray.push(ELPInfo({
              walletAddress: _elpAddress,
              index: index,
              allocPoint: _point,
              statusActive : true}));
      }
  }

  function addeteamAddress(address _eteamAddress) public onlyOwner{
      if(eteamAddress[_eteamAddress]==0){
          uint index = eteamArray.length;
          if(eteamArray.length == 0){
              index = 1;
          }
          eteamAddress[_eteamAddress] = index+1;
          eteamArray.push(UserInfo({
              walletAddress: _eteamAddress,
              index: index,
              statusActive: true}));
      }
  }
```

```
// Disabled adv address from advArray by setting statusActive to false
    function removeAdvAddress(address _advAddress) public onlyOwner{
         if(advAddress[_advAddress]>0){
             for(uint i = 0; i < advArray.length ; i++){
                 if(advArray[i].walletAddress == _advAddress){
                    advArray[i].statusActive = false;
                 }
             }
         }
    }

    // Disabled eteam address from eteamArray by setting statusActive to false
    function removeEteamAddress(address _eteamAddress) public onlyOwner{
          if(eteamAddress[_eteamAddress]>0){
             for(uint i = 0; i < eteamArray.length ; i++){
                 if(eteamArray[i].walletAddress == _eteamAddress){
                    eteamArray[i].statusActive = false;
                 }
             }
         }
    }

    // Disabled eteam address from eteamArray by setting statusActive to false
    function removElpAddress(address _elpAddress) public onlyOwner{
          if(elpAddress[_elpAddress]>0){
             for(uint i = 0; i < elpArray.length ; i++){
                 if(elpArray[i].walletAddress == _elpAddress){
                    elpArray[i].statusActive = false;
                 }
             }
         }
    }

    function changeStakeHolderFundTimeLock(address _stakeHolderFundTimelock) public onlyOwner{
       stakeHolderFundTimelock = _stakeHolderFundTimelock;
    }

}
```

Testing

Primary checks followed during testing of Smart Contract is to see that if code :

- We check the Smart Contracts Logic and compare it with the standards, check the gas usage optimization and the implemented business logic.
- The contract code should follow the Conditions and logic as per user request.
- We deploy the Contract and run the Tests.
- We make sure that the Contract does not lose any money/Ether.

Known Vulnerabilities Check

Smart Contract was scanned for commonly known and more specific vulnerabilities.
Following are the list of commonly known vulnerabilities that was considered during the (manual and automated) audit of the smart contract and their comments:

1. TimeStamp Dependence : The timestamp of the block can be manipulated by the miner, and so should not be used for critical components of the contract. If required, *Block numbers* and *average block time* can be used to estimate time.

Comments : Kingswap smart contracts do not have any timestamp dependence in their code.

2. Gas Limit and Loops : Sometimes the number of iterations in a loop can cause the transaction gas limit to grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Hence, the loops must be used in a very calculative fashion.

Comments : Kingswap smart contracts are free from the gas limit check as the contracts do not contain any loop in their code.

3. Compiler Version : Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed to an upgraded compiler version, as future compiler versions may handle certain language constructions in a way the developer did not foresee.

Comments : In Kingswap smart contracts the compiler version is fixed to 0.6.12.

4. ERC20 Standards : KingToken smart contract follows all the universal ERC20 coding standards and implements all its functions and events in the contract code.

5. Redundant fallback function : The standard execution cost of a fallback function should be less than 2300 gas.

Comments : Kingswap smart contracts do not have any fallback function, hence they are free from this vulnerability.

6. Unchecked math : Need to guard uint overflow or security flaws by implementing the proper maths logic checks.

Comments: The Kingswap smart contracts use the popular SafeMath library for critical operations to avoid arithmetic over or underflow and safeguard against unwanted behaviour. In particular, the balances variable is updated using the safemath operation.

7. Exception disorder : When an exception is thrown, it cannot be caught: the execution stops, the fee is lost. The irregularity in how exceptions are handled may affect the security of contracts.

Comments : In manual testing of Kingswap smart contracts, there are no exception disorders found due to code or syntax issues.

8. Reentrancy : The reentrancy attack consists of the recursively calling a method to extract ether from a contract if the user is not updating the balance of the sender before sending the ether.

Comments : There is no risk of reentrancy attack in Kingswap smart contracts as there are no calls to external functions for sending ether.

9. DoS with (Unexpected) Throw : The Contract code can be vulnerable to the Call Depth Attack! So instead, code should have a pull payment system instead of push.

Comments : The Kingswap smart contracts do not implement any payment related scenario thus not vulnerable to this attack.

10. DoS with Block Gas Limit : In a contract by paying out to everyone at once, contract risk running into the block gas limit. Each ethereum block can process a certain maximum amount of computation. If one tries to go over that, the transaction will fail. Therefore again push over pull payment is suggested to remove the above issue.

Comments : The Kingswap smart contracts do not implement any payment related scenario thus not vulnerable to this attack.

11. Explicit Visibility in functions and state variables : Explicit visibility in the function and state variables are provided. Visibility like external, internal, private and public is used and defined properly.

Comments : There are variables in the Kingswap smart contracts which are declared with "private" modifiers. We would like to point out a popular misconception among developers that "private" modifiers make a variable invisible to the outside world (of smart contract). However, the miners or investors have view access to all of the contract's code, data or state changes.

12. **Using the approve function of ERC20 standard :** There has been a vulnerability in using the approve function of ERC20 standard. Let's take an example, suppose an

address A approves an address B to spend 100 tokens. Later, A decides to approve B to spend 50 more tokens i.e. 150 tokens. If B is monitoring pending transactions, when B sees A's new approval, B can attempt to quickly spend 100 tokens, racing to get his transaction mined prior to A's new approval being written to the blockchain. If B's transaction beats A's, then B can spend another 150 tokens after A's transaction for approval of 150 tokens.

This issue is a consequence of the ERC20 standard, which specifies that `approve()`takes a replacement value, but no prior value. Preventing the attack while complying with ERC20 involves a compromise, The users should set the approval to zero, make sure B hasn't snuck in a spend, and then set the new value. This will increase an extra transaction, to reduce the approved amount to zero.

Comments : This issue is mitigated in KingToken smart contract, by adding "increaseAllowance" and "decreaseAllowance" functions.

13. Short address attack : Recently, the Golem team discovered that an exchange wasn't validating user-entered addresses on transfers. Due to the way `msg.data` is interpreted, it was possible to enter a shortened address, which would cause the server to construct a transfer transaction that would appear correct to server-side code, but would actually transfer a much larger amount than expected.

Comments : Vulnerability level : moderate

In KingToken smart contract there is no check on address length, this attack can be entirely prevented by doing a length check on `msg.data`. In the case of `transfer()`, the length should be 68. Consider the example modules below as a hint on how to implement short address attach check:

```
modifier onlyPayloadSize(uint size) {

assert(msg.data.length == size + 4); //length must be 68

_;}

function transfer(address _to, uint256 _value) onlyPayloadSize(2 * 32) { // do stuff

}
```

Automated test results

We have used the truffle framework and smart check tool for writing and generating automated test results.

Below are the ERC20 test cases checked :

- Should have correct "name" definition - passed

- Should have correct "totalSupply" definition - passed - Should have correct "symbol" definition - passed

- Should have correct "decimals" definition - passed

- Should have correct "balanceOf" definition - passed

- Should have correct "transfer" definition - passed

- Should have correct "transferFrom" definition - passed

- Should have correct "approve" definition - passed

- Should have correct "allowance" definition - passed

- Should have correct "Transfer" definition - passed

- Should have correct "Approval" definition - passed

- The deployer of the contract is the owner of the contract - passed

- Should generate an ERC20 token with proper configuration - passed

- Owner must be allocated tokens as per the minting function called in the constructor - passed - Owner address is able to transfer tokens to other addresses - passed

- Two randomly generated ethereum addresses are able to receive tokens and transfer - passed

- Should throw error on trying to transfer negative token amount - passed

- An address cannot transfer tokens more than it has - passed

- Should allow an address to approve another address for transferring tokens - passed

- Should allow an address to zero out the previously allowed amount - passed

- Should not allow any address to send tokens to 0x0 address - passed

- Should not allow spender address to send more tokens than it has been approved to send - passed

- Should allow an approval to be set, increased and decreased - passed

SafeMath Library

- Should skip operation on multiply by zero - passed - Should revert on multiply overflow - passed

- Should revert on multiply by zero - passed

- Should allow regular multiply - passed

- Should allow regular division - passed

- Should revert on subtraction overflow - passed - Should revert on addition overflow - passed

- Should allow regular subtraction - passed

- Should allow regular addition - passed

AUTOMATED TEST RESULTS -

Passed - 32

Failed - 0

Risk

NO CRITICAL RISKS FOUND

The Kingswap Smart Contracts have no risk of losing any amounts of ether/tokens in case of external attack or a bug, as the contracts do not take any kind of funds from the user as investment. If anyone tries to send any amount of ether to the contract address, the transaction will cancel itself and no ether comes to the contracts.

Conclusion and Results

In this report, we have concluded about the security standards of Kingswap Smart Contracts. The smart contract has been analysed under different facets. Code quality is very good, and well modularised. We found that Kingswap smart contract adapts a very good coding practice and has clean, documented code. No severe discrepancies were found.