Armors Labs

HaykerDAO (HKR Yield Farming)

Smart Contract Audit

HaykerDAO (HKR Yield Farming) Audit Summary

Project name : HaykerDAO (HKR Yield Farming) Contract

Project address: None

Code URL : https://github.com/HaykerDAO/mining

Commit : 7191e4e182dbac184ccd03b04558d488023f302c

Project target : HaykerDAO (HKR Yield Farming) Contract Audit

Blockchain : Huobi ECO Chain（Heco）

Test result : PASSED

Audit Info

Audit NO : 0X202105110008

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

HaykerDAO (HKR Yield Farming) Audit

The HaykerDAO team asked us to review and audit their HaykerDAO (HKR Yield Farming) contract. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
HaykerDAO (HKR Yield Farming) Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-05-11

Audit results

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the HaykerDAO (HKR Yield Farming) contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 3 months from the date of output.

(Statement: Armors Labs reports only on facts that have occurred or existed before this report is issued and assumes corresponding responsibilities. Armors Labs is not able to determine the security of its smart contracts and is not responsible for any subsequent or existing facts after this report is issued. The security audit analysis and other content of this report are only based on the documents and information provided by the information provider to Armors Labs at the time of issuance of this report (" information provided " for short). Armors Labs postulates that the information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered, deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused.)

Audited target file

file	md5
./contracts/MasterChef.sol	4f9e2c53ad266943c663a6332262f68c
./contracts/TokenAmountFromDex.sol	93097266745989e97062c01000302eb2

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe

Vulnerability	status
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

```
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Affero General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Affero General Public License for more details.

pragma solidity 0.6.12;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     *
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
```

```
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     *
     * - Multiplication cannot overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}
/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }
    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.co
        return msg.data;
    }
}

// File: @openzeppelin/contracts/access/Ownable.sol

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

interface DexPairLike {
    function sync()        external;
    function token0()      external view returns (address);
    function token1()      external view returns (address);
    function getReserves() external view returns (uint112, uint112, uint32);  // reserve0, reserve1,
}

contract TokenAmountFromDex is Ownable {
    using SafeMath for uint256;

    DexPairLike public haiHkrPairContract = DexPairLike(0x208B5ADd7918db2E36811436CEe87deEF1dA73C2);
    address public HAI_ADDRESS = 0x7663Bc3Ae9858cae71722aeDeE364E125C278bdf;
    address public HKR_ADDRESS = 0xA74b0514B403bdb573BF22dF0062d43F6498a164;

    event SetHaiHkrPairContract(DexPairLike haiHkrPairContract);

    // Update the harvest fee ratio
    function setHaiHkrPairContract(DexPairLike _haiHkrPairContract) public onlyOwner {
        require(_haiHkrPairContract != DexPairLike(0), "haiHkrPairContract can not be zero!");
        haiHkrPairContract = _haiHkrPairContract;
        emit SetHaiHkrPairContract(_haiHkrPairContract);
    }

    // token amount = hkr amount X hkr price / _token price
    function getTokenAmount(address _token, uint256 hkrAmount) external view returns (uint256){
        require(_token == HAI_ADDRESS, "Only support HAI for now!");
        (uint112 haiReserve, uint112 hkrReserve, uint32 blockTimestampLast) = haiHkrPairContract.getR
        if(hkrReserve > 0){
            return hkrAmount.mul(haiReserve).div(hkrReserve);
        }
        return 0;
    }
}


// File: @openzeppelin/contracts/token/ERC20/IERC20.sol


pragma solidity 0.6.12;

/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
interface IERC20 {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Returns the remaining number of tokens that `spender` will be
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowance(address owner, address spender) external view returns (uint256);

    /**
     * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * transaction ordering. One possible solution to mitigate this race
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 amount) external returns (bool);

    /**
     * @dev Moves `amount` tokens from `sender` to `recipient` using the
     * allowance mechanism. `amount` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}

// File: @openzeppelin/contracts/math/SafeMath.sol

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     *
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     *
     * - Multiplication cannot overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}


/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.co
        return msg.data;
    }
}

// File: @openzeppelin/contracts/access/Ownable.sol

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

// token amount = sushi amount X sushi price / _token price
interface TokenAmountLike {
    function getTokenAmount(address _token, uint256 hkrAmount) external view returns (uint256);
}

/**
 * @title Helps contracts guard agains rentrancy attacks.
 * @author Remco Bloemen <remco@2π.com>
 * @notice If you mark a function `nonReentrant`, you should also
 * mark it `external`.
 */
contract ReentrancyGuard {

  /**
   * @dev We use a single lock for the whole contract.
   */
  bool private rentrancy_lock = false;

  /**
   * @dev Prevents a contract from calling itself, directly or indirectly.
   * @notice If you mark a function `nonReentrant`, you should also
   * mark it `external`. Calling one nonReentrant function from
   * another is not supported. Instead, you can implement a
   * `private` function doing the actual work, and a `external`
   * wrapper marked as `nonReentrant`.
   */
  modifier nonReentrant() {
    require(!rentrancy_lock);
    rentrancy_lock = true;
    _;
```

```
      rentrancy_lock = false;
    }

}

// MasterChef is the master of Sushi. He can make Sushi 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 SUSHI is sufficiently
// distributed and the community can show to govern itself.
//
// Have fun reading it. Hopefully it's bug-free. God bless.

contract MasterChef is Ownable, ReentrancyGuard{
    using SafeMath for uint256;

    // 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 SUSHIs
        // entitled to a user but is pending to be distributed is:
        //
        //   pending reward = (user.amount * pool.accSushiPerShare) - user.rewardDebt
        //
        // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
        //   1. The pool's `accSushiPerShare` (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, zero represents HT pool.
        uint256 amount;     // How many LP tokens the pool has.
        uint256 rewardForEachBlock;    //Reward for each block
        uint256 lastRewardBlock;  // Last block number that SUSHIs distribution occurs.
        uint256 accSushiPerShare; // Accumulated SUSHIs per share, times 1e12. See below.
        uint256 startBlock; // Reward start block.
        uint256 endBlock;  // Reward end block.
        uint256 rewarded;// the total sushi has beed reward, including the dev and user harvest

        uint256 operationFee;// Charged when user operate the pool, only deposit firstly.
        address operationFeeToken;// empty reprsents charged with mainnet token.

        uint16 harvestFeeRatio;// Charged when harvest, div RATIO_BASE for the real ratio, like 100 f
        address harvestFeeToken;// empty reprsents charged with mainnet token.
    }

    uint256 private constant ACC_SUSHI_PRECISION = 1e12;

    uint8 public constant ZERO = 0 ;
    uint16 public constant RATIO_BASE = 1000;

    uint8 public constant DEV1_SUSHI_REWARD_RATIO = 68;// div RATIO_BASE
    uint8 public constant DEV2_SUSHI_REWARD_RATIO = 48;// div RATIO_BASE
    uint8 public constant DEV3_SUSHI_REWARD_RATIO = 34;// div RATIO_BASE
    uint16 public constant MINT_SUSHI_REWARD_RATIO = 850;// div RATIO_BASE

    uint16 public constant DEV1_FEE_RATIO = 500;// div RATIO_BASE
    uint16 public constant DEV2_FEE_RATIO = 250;// div RATIO_BASE
    uint16 public constant DEV3_FEE_RATIO = 250;// div RATIO_BASE

    uint16 public harvestFeeBuyRatio = 800;// the buy ratio for harvest, div RATIO_BASE
    uint16 public harvestFeeDevRatio = 200;// the dev ratio for harvest, div RATIO_BASE

    // The SUSHI TOKEN!
    IERC20 public sushi;
    // Dev address.
    address payable public dev1Address;
    address payable public dev2Address;
    address payable public dev3Address;

    address payable public buyAddress;// address for the fee to buy HKR

    TokenAmountLike public tokenAmountContract;

    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping (uint256 => mapping (address => UserInfo)) public userInfo;

    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);
    event Harvest(address indexed user, uint256 indexed pid, uint256 amount);
    event EmergencyStop(address indexed user, address to);
    event Add(uint256 rewardForEachBlock, IERC20 lpToken, bool withUpdate,
    uint256 startBlock, uint256 endBlock, uint256 operationFee, address operationFeeToken,
    uint16 harvestFeeRatio, address harvestFeeToken, bool _withSushiTransfer);
    event SetPoolInfo(uint256 pid, uint256 rewardsOneBlock, bool withUpdate, uint256 startBlock, uint
    event ClosePool(uint256 pid, address payable to);
    event UpdateDev1Address(address payable dev1Address);
    event UpdateDev2Address(address payable dev2Address);
    event UpdateDev3Address(address payable dev3Address);
    event UpdateBuyAddress(address payable buyAddress);
    event AddRewardForPool(uint256 pid, uint256 _addSushiPerPool, uint256 _addSushiPerBlock, bool wit

    event SetPoolOperationFee(uint256 pid, uint256 operationFee, address operationFeeToken, bool feeU
    event SetPoolHarvestFee(uint256 pid, uint16 harvestFeeRatio, address harvestFeeToken, bool feeRat

    event SetTokenAmountContract(TokenAmountLike tokenAmountContract);

    event SetHarvestFeeRatio(uint16 harvestFeeBuyRatio, uint16 harvestFeeDevRatio);

    constructor(
        IERC20 _sushi,
        address payable _dev1Address,
        address payable _dev2Address,
        address payable _dev3Address,
        address payable _buyAddress,
        TokenAmountLike _tokenAmountContract
    ) public {
        sushi = _sushi;
        dev1Address = _dev1Address;
        dev2Address = _dev2Address;
        dev3Address = _dev3Address;
        buyAddress = _buyAddress;
        tokenAmountContract = _tokenAmountContract;
    }

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

    function setTokenAmountContract(TokenAmountLike _tokenAmountContract) public onlyOwner {
        require(_tokenAmountContract != TokenAmountLike(ZERO), "tokenAmountContract can not be zero!"
        tokenAmountContract = _tokenAmountContract;
        emit SetTokenAmountContract(_tokenAmountContract);
    }
```

```
// Update the harvest fee ratio
function setHarvestFeeRatio(uint16 _harvestFeeBuyRatio, uint16 _harvestFeeDevRatio) public onlyOw
    require((_harvestFeeBuyRatio + _harvestFeeDevRatio) == RATIO_BASE, "The sum must be 1000!");
    harvestFeeBuyRatio = _harvestFeeBuyRatio;
    harvestFeeDevRatio = _harvestFeeDevRatio;
    emit SetHarvestFeeRatio(_harvestFeeBuyRatio, _harvestFeeDevRatio);
}

// Add a new lp to the pool. Can only be called by the owner.
// Zero lpToken represents HT pool.
function add(uint256 _rewardForEachBlock, IERC20 _lpToken, bool _withUpdate,
uint256 _startBlock, uint256 _endBlock, uint256 _operationFee, address _operationFeeToken,
uint16 _harvestFeeRatio, address _harvestFeeToken, bool _withSushiTransfer) public onlyOwner {
    //require(_lpToken != IERC20(ZERO), "lpToken can not be zero!");
    require(_rewardForEachBlock > ZERO, "rewardForEachBlock must be greater than zero!");
    require(_startBlock < _endBlock, "start block must less than end block!");
    if (_withUpdate) {
        massUpdatePools();
    }
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        amount: ZERO,
        rewardForEachBlock: _rewardForEachBlock,
        lastRewardBlock: block.number > _startBlock ? block.number : _startBlock,
        accSushiPerShare: ZERO,
        startBlock: _startBlock,
        endBlock: _endBlock,
        rewarded: ZERO,
        operationFee: _operationFee,
        operationFeeToken: _operationFeeToken,
        harvestFeeRatio: _harvestFeeRatio,
        harvestFeeToken: _harvestFeeToken
    }));
    if(_withSushiTransfer){
        uint256 amount = (_endBlock - (block.number > _startBlock ? block.number : _startBlock)).
        sushi.transferFrom(msg.sender, address(this), amount);
    }
    emit Add(_rewardForEachBlock, _lpToken, _withUpdate, _startBlock, _endBlock, _operationFee, _
}

// Update the given pool's pool info. Can only be called by the owner.
function setPoolInfo(uint256 _pid, uint256 _rewardForEachBlock, bool _withUpdate, uint256 _startB
    if (_withUpdate) {
        massUpdatePools();
    }
    PoolInfo storage pool = poolInfo[_pid];
    if(_startBlock > ZERO){
        if(_endBlock > ZERO){
            require(_startBlock < _endBlock, "start block must less than end block!");
        }else{
            require(_startBlock < pool.endBlock, "start block must less than end block!");
        }
        pool.startBlock = _startBlock;
    }
    if(_endBlock > ZERO){
        if(_startBlock <= ZERO){
            require(pool.startBlock < _endBlock, "start block must less than end block!");
        }
        pool.endBlock = _endBlock;
    }
    if(_rewardForEachBlock > ZERO){
        pool.rewardForEachBlock = _rewardForEachBlock;
    }
    emit SetPoolInfo(_pid, _rewardForEachBlock, _withUpdate, _startBlock, _endBlock);
}
```

```
function setAllPoolOperationFee(uint256 _operationFee, address _operationFeeToken, bool _feeUpdat
    uint256 length = poolInfo.length;
    for (uint256 pid = ZERO; pid < length; ++ pid) {
        setPoolOperationFee(pid, _operationFee, _operationFeeToken, _feeUpdate, _feeTokenUpdate);
    }
}

// Update the given pool's operation fee
function setPoolOperationFee(uint256 _pid, uint256 _operationFee, address _operationFeeToken, boo
    updatePool(_pid);
    PoolInfo storage pool = poolInfo[_pid];
    if(_feeUpdate){
        pool.operationFee = _operationFee;
    }
    if(_feeTokenUpdate){
        pool.operationFeeToken = _operationFeeToken;
    }
    emit SetPoolOperationFee(_pid, _operationFee, _operationFeeToken, _feeUpdate, _feeTokenUpdate
}

function setAllPoolHarvestFee(uint16 _harvestFeeRatio, address _harvestFeeToken, bool _feeRatioUp
    uint256 length = poolInfo.length;
    for (uint256 pid = ZERO; pid < length; ++ pid) {
        setPoolHarvestFee(pid, _harvestFeeRatio, _harvestFeeToken, _feeRatioUpdate, _feeTokenUpda
    }
}

// Update the given pool's harvest fee
function setPoolHarvestFee(uint256 _pid, uint16 _harvestFeeRatio, address _harvestFeeToken, bool 
    updatePool(_pid);
    PoolInfo storage pool = poolInfo[_pid];
    if(_feeRatioUpdate){
        pool.harvestFeeRatio = _harvestFeeRatio;
    }

    if(_feeTokenUpdate){
        pool.harvestFeeToken = _harvestFeeToken;
    }
    emit SetPoolHarvestFee(_pid, _harvestFeeRatio, _harvestFeeToken, _feeRatioUpdate, _feeTokenUp
}

// Return reward multiplier over the given _from to _to block.
function getMultiplier(uint256 _from, uint256 _to) public pure returns (uint256) {
    if(_to > _from){
        return _to.sub(_from);
    }
    return ZERO;
}

// 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;
    }
    if (block.number < pool.startBlock){
        return;
    }
    if (pool.lastRewardBlock >= pool.endBlock){
         return;
    }
    if (pool.lastRewardBlock < pool.startBlock) {
        pool.lastRewardBlock = pool.startBlock;
    }
    uint256 multiplier;
```

```
        if (block.number > pool.endBlock){
            multiplier = getMultiplier(pool.lastRewardBlock, pool.endBlock);
            pool.lastRewardBlock = pool.endBlock;
        }else{
            multiplier = getMultiplier(pool.lastRewardBlock, block.number);
            pool.lastRewardBlock = block.number;
        }
        uint256 lpSupply = pool.amount;
        if (lpSupply <= ZERO) {
            return;
        }
        uint256 sushiReward = multiplier.mul(pool.rewardForEachBlock);
        if(sushiReward > ZERO){
            transferToDev(pool, dev1Address, DEV1_SUSHI_REWARD_RATIO, sushiReward);
            transferToDev(pool, dev2Address, DEV2_SUSHI_REWARD_RATIO, sushiReward);
            transferToDev(pool, dev3Address, DEV3_SUSHI_REWARD_RATIO, sushiReward);
            uint256 poolSushiReward = sushiReward.mul(MINT_SUSHI_REWARD_RATIO).div(RATIO_BASE);
            pool.accSushiPerShare = pool.accSushiPerShare.add(poolSushiReward.mul(ACC_SUSHI_PRECISION
        }
    }

    function transferToDev(PoolInfo storage _pool, address _devAddress, uint16 _devRatio, uint256 _su
        amount = _sushiReward.mul(_devRatio).div(RATIO_BASE);
        safeTransferTokenFromThis(sushi, _devAddress, amount);
        _pool.rewarded = _pool.rewarded.add(amount);
    }

    // View function to see pending SUSHIs on frontend.
    function pendingSushi(uint256 _pid, address _user) public view returns (uint256 sushiReward, uint
        PoolInfo storage pool =  poolInfo[_pid];
        if(_user == address(ZERO)){
            _user = msg.sender;
        }
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accSushiPerShare = pool.accSushiPerShare;
        uint256 lpSupply = pool.amount;
        uint256 lastRewardBlock = pool.lastRewardBlock;
        if (lastRewardBlock < pool.startBlock) {
            lastRewardBlock = pool.startBlock;
        }
        if (block.number > lastRewardBlock && block.number >= pool.startBlock && lastRewardBlock < po
            uint256 multiplier = ZERO;
            if (block.number > pool.endBlock){
                multiplier = getMultiplier(lastRewardBlock, pool.endBlock);
            }else{
                multiplier = getMultiplier(lastRewardBlock, block.number);
            }
            uint256 poolSushiReward = multiplier.mul(pool.rewardForEachBlock).mul(MINT_SUSHI_REWARD_R
            accSushiPerShare = accSushiPerShare.add(poolSushiReward.mul(ACC_SUSHI_PRECISION).div(lpSu
        }
        sushiReward = user.amount.mul(accSushiPerShare).div(ACC_SUSHI_PRECISION).sub(user.rewardDebt)

        fee = getHarvestFee(pool, sushiReward);
    }

    function getHarvestFee(PoolInfo storage _pool, uint256 _sushiAmount) private view returns (uint25
        uint256 fee = ZERO;
        if(_pool.harvestFeeRatio > ZERO && tokenAmountContract != TokenAmountLike(ZERO)){//charge for
            fee = tokenAmountContract.getTokenAmount(_pool.harvestFeeToken, _sushiAmount).mul(_pool.h
        }
        return fee;
    }

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

    // Deposit LP tokens to MasterChef for SUSHI allocation.
    function deposit(uint256 _pid, uint256 _amount) public payable {
        PoolInfo storage pool = poolInfo[_pid];
        require(block.number <= pool.endBlock, "this pool is end!");
        require(block.number >= pool.startBlock, "this pool is not start!");
        if(pool.lpToken == IERC20(0)){//if pool is HT
            require((_amount + pool.operationFee) == msg.value, "msg.value must be equals to amount +
        }
        checkOperationFee(pool, _amount);
        UserInfo storage user = userInfo[_pid][msg.sender];
        harvest(_pid, msg.sender);
        if(pool.lpToken != IERC20(0)){
            pool.lpToken.transferFrom(msg.sender, address(this), _amount);
        }
        pool.amount = pool.amount.add(_amount);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(ACC_SUSHI_PRECISION);
        emit Deposit(msg.sender, _pid, _amount);
    }

    function checkOperationFee(PoolInfo storage _pool, uint256 _amount) private nonReentrant {
        if(_pool.operationFee > ZERO){// charge for fee
            uint256 dev1Amount = _pool.operationFee.mul(DEV1_FEE_RATIO).div(RATIO_BASE);
            uint256 dev2Amount = _pool.operationFee.mul(DEV2_FEE_RATIO).div(RATIO_BASE);
            uint256 dev3Amount = _pool.operationFee.sub(dev1Amount).sub(dev2Amount);
            if(isMainnetToken(_pool.operationFeeToken)){
                if(_pool.lpToken != IERC20(0)){
                    require(msg.value == _pool.operationFee, "Fee is not enough or too much!");
                }else{//if pool is HT
                    require((msg.value - _amount) == _pool.operationFee, "Fee is not enough or too mu
                }
                dev1Address.transfer(dev1Amount);
                dev2Address.transfer(dev2Amount);
                dev3Address.transfer(dev3Amount);
            }else{
                IERC20 token = IERC20(_pool.operationFeeToken);
                uint feeBalance = token.balanceOf(msg.sender);
                require(feeBalance >= _pool.operationFee, "Fee is not enough!");

                token.transferFrom(msg.sender, address(this), _pool.operationFee);

                token.transfer(dev1Address, dev1Amount);
                token.transfer(dev2Address, dev2Amount);
                token.transfer(dev3Address, dev3Amount);
            }
        }
    }

    function isMainnetToken(address _token) private pure returns (bool) {
        return _token == address(ZERO);
    }

    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public payable {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(block.number >= pool.startBlock,"this pool is not start!");
        require(user.amount >= _amount, "withdraw: not good");
        harvest(_pid, msg.sender);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(ACC_SUSHI_PRECISION);
        if(pool.lpToken != IERC20(0)){
            pool.lpToken.transfer(msg.sender, _amount);
        }else{//if pool is HT
            transferMainnetToken(msg.sender, _amount);
        }
        pool.amount = pool.amount.sub(_amount);
        emit Withdraw(msg.sender, _pid, _amount);
    }

    //transfer HT
    function transferMainnetToken(address payable _to, uint256 _amount) private nonReentrant {
        _to.transfer(_amount);
    }

    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        if(pool.lpToken != IERC20(0)){
            pool.lpToken.transfer(msg.sender, user.amount);
        }else{//if pool is HT
            transferMainnetToken(msg.sender, user.amount);
        }
        pool.amount = pool.amount.sub(user.amount);
        uint256 oldAmount = user.amount;
        user.amount = ZERO;
        user.rewardDebt = ZERO;
        emit EmergencyWithdraw(msg.sender, _pid, oldAmount);
    }

    function harvest(uint256 _pid, address _to) public nonReentrant payable returns (bool success) {
        if(_to == address(ZERO)){
            _to = msg.sender;
        }
        PoolInfo storage pool =  poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_to];
        updatePool(_pid);
        uint256 pending = user.amount.mul(pool.accSushiPerShare).div(ACC_SUSHI_PRECISION).sub(user.re
        if (pending > ZERO) {
            success = true;
            checkHarvestFee(pool, pending);
            safeTransferTokenFromThis(sushi, _to, pending);
            pool.rewarded = pool.rewarded.add(pending);
            user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(ACC_SUSHI_PRECISION);
        } else{
            success = false;
        }
        emit Harvest(_to, _pid, pending);
    }

    function checkHarvestFee(PoolInfo storage _pool, uint256 _sushiReward) private {
        uint256 fee = getHarvestFee(_pool, _sushiReward);
        if(fee > ZERO){
            uint256 devFee = fee.mul(harvestFeeDevRatio).div(RATIO_BASE);
            uint256 buyFee = fee.sub(devFee);

            uint256 dev1Amount = devFee.mul(DEV1_FEE_RATIO).div(RATIO_BASE);
            uint256 dev2Amount = devFee.mul(DEV2_FEE_RATIO).div(RATIO_BASE);
            uint256 dev3Amount = devFee.sub(dev1Amount).sub(dev2Amount);

            if(isMainnetToken(_pool.harvestFeeToken)){
                require(msg.value == fee, "Fee is not enough or too much!");
                dev1Address.transfer(dev1Amount);
                dev2Address.transfer(dev2Amount);
                dev3Address.transfer(dev3Amount);
                buyAddress.transfer(buyFee);
            }else{
                IERC20 token = IERC20(_pool.harvestFeeToken);
                uint feeBalance = token.balanceOf(msg.sender);
                require(feeBalance >= fee, "Fee is not enough!");
                token.transferFrom(msg.sender, address(this), fee);

                token.transfer(dev1Address, dev1Amount);
                token.transfer(dev2Address, dev2Amount);
                token.transfer(dev3Address, dev3Amount);
                token.transfer(buyAddress, buyFee);
            }
        }
    }

    function emergencyStop(address payable _to) public onlyOwner {
        if(_to == address(ZERO)){
            _to = msg.sender;
        }
        uint addrBalance = sushi.balanceOf(address(this));
        if(addrBalance > ZERO){
            sushi.transfer(_to, addrBalance);
        }
        uint256 length = poolInfo.length;
        for (uint256 pid = ZERO; pid < length; ++ pid) {
            closePool(pid, _to);
        }
        emit EmergencyStop(msg.sender, _to);
    }

    function closePool(uint256 _pid, address payable _to) public onlyOwner {
        PoolInfo storage pool = poolInfo[_pid];
        pool.endBlock = block.number;
        if(_to == address(ZERO)){
            _to = msg.sender;
        }
        emit ClosePool(_pid, _to);
    }

    // Safe transfer token function, just in case if rounding error causes pool to not have enough to
    function safeTransferTokenFromThis(IERC20 _token, address _to, uint256 _amount) internal {
        uint256 bal = _token.balanceOf(address(this));
        if (_amount > bal) {
            _token.transfer(_to, bal);
        } else {
            _token.transfer(_to, _amount);
        }
    }

     // Update dev1 address by the previous dev.
    function updateDev1Address(address payable _dev1Address) public {
        require(msg.sender == dev1Address, "dev1: wut?");
        require(_dev1Address != address(ZERO), "address can not be zero!");
        dev1Address = _dev1Address;
        emit UpdateDev1Address(_dev1Address);
    }

    // Update dev2 address by the previous dev.
    function updateDev2Address(address payable _dev2Address) public {
        require(msg.sender == dev2Address, "dev2: wut?");
        require(_dev2Address != address(ZERO), "address can not be zero!");
        dev2Address = _dev2Address;
        emit UpdateDev2Address(_dev2Address);
    }

    // Update dev3 address by the previous dev.
    function updateDev3Address(address payable _dev3Address) public {
        require(msg.sender == dev3Address, "dev3: wut?");
        require(_dev3Address != address(ZERO), "address can not be zero!");
        dev3Address = _dev3Address;
        emit UpdateDev3Address(_dev3Address);
    }

    // Update dev3 address by the previous dev.
    function updateBuyAddress(address payable _buyAddress) public {
        require(msg.sender == buyAddress, "buyAddress: wut?");
        require(_buyAddress != address(ZERO), "address can not be zero!");
        buyAddress = _buyAddress;
        emit UpdateBuyAddress(_buyAddress);
    }

    // Add reward for pool from the current block or start block
    function addRewardForPool(uint256 _pid, uint256 _addSushiPerPool, uint256 _addSushiPerBlock, bool
        require(_addSushiPerPool > ZERO || _addSushiPerBlock > ZERO, "add sushi must be greater than
        PoolInfo storage pool = poolInfo[_pid];
        require(block.number < pool.endBlock, "this pool is going to be end or end!");
        updatePool(_pid);
        uint256 addSushiPerBlock = _addSushiPerBlock;
        uint256 addSushiPerPool = _addSushiPerPool;
        uint256 start = block.number;
        uint256 end = pool.endBlock;
        if(start < pool.startBlock){
            start = pool.startBlock;
        }
        uint256 blockNumber = end.sub(start);
        if(blockNumber <= ZERO){
            blockNumber = 1;
        }
        if(addSushiPerBlock <= ZERO){
            addSushiPerBlock = _addSushiPerPool.div(blockNumber);
        }
        addSushiPerPool = addSushiPerBlock.mul(blockNumber);
        pool.rewardForEachBlock = pool.rewardForEachBlock.add(addSushiPerBlock);
        if(_withSushiTransfer){
            sushi.transferFrom(msg.sender, address(this), addSushiPerPool);
        }
        emit AddRewardForPool(_pid, addSushiPerPool, addSushiPerBlock, _withSushiTransfer);
    }
}
```

Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unexpected Blockchain Currency

- **Description:**
  Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Delegatecall

- **Description:**
  The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
  Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whBlockchain Currency a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifers can lead to some devestating vulernabilities in smart contracts as will be discussed in this section.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Entropy Illusion

- **Description:**
  All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

External Contract Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
  The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Denial Of Service (DOS)

- **Description:**
  This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Block Timestamp Manipulation

- **Description:**
  Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Constructors with Care

- **Description:**
  Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unintialised Storage Pointers

- **Description:**
  The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

tx.origin Authentication

- **Description:**
  Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

armors.io

contact@armors.io