Armors Labs

EarthFundBonding

Smart Contract Audit

EarthFundBonding Audit Summary

Project name : EarthFundBonding Contract

Project address: None

Code URL : https://github.com/earthfund-io/earthfund-contracts/blob/master/contracts/EarthFundBonding.sol

Commit : 44e7b5cc3eaafcdf571cfdbf655f19007008a1f6

Project target : EarthFundBonding Contract Audit

Blockchain : Ethereum

Test result : PASSED

Audit Info

Audit NO : 0X202203250016

Audit Team : Armors Labs

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

EarthFundBonding Audit

The EarthFundBonding team asked us to review and audit their EarthFundBonding 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
EarthFundBonding Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2022-03-25

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 EarthFundBonding 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.

Disclaimer

Armors Labs Reports is not and should not be regarded as an "approval" or "disapproval" of any particular project or team. These reports are not and should not be regarded as indicators of the economy or value of any "product" or "asset" created by any team. Armors do not cover testing or auditing the integration with external contract or services (such as Unicrypt, Uniswap, PancakeSwap etc'…)

Armors Labs Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Armors does not guarantee the safety or functionality of the technology agreed to be analyzed.

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

Audited target file

file	md5
EarthFundBonding.sol	57fca99c61d5792641b7ca11ede75408

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

Vulnerability	status
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

EarthFundBonding.sol

```
// SPDX-License-Identifier: AGPL-3.0-or-later
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/math/Math.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

/*
 * 1EARTH bonding contract
 * This contract allows users to "bond" USDT or ETH for the 1EARTH token. This means that
 * users can purchase 1EARTH at a variable rate with either USDT or ETH, and the resulting
 * token vests over a predetermined amount of time. Users are then able to claim these tokens.
 *
 * Funds received from this contract are sent to the EarthFund DAO Treasury, where DAO members will
 * will vote as to where they are distributed
 */
contract EarthFundBonding is Ownable {
    using SafeERC20 for IERC20;
    using SafeMath for uint256;

    /* ======== EVENTS ======== */

    event BondCreated(
        uint256 deposit,
        uint256 indexed payout,
        uint256 indexed expires,
        uint256 indexed price
    );

    event BondRedeemed(
        address indexed recipient,
        uint256 payout,
        uint256 remaining
    );

    event BondPriceChanged(
        uint256 indexed bondIndex,
        uint256 indexed internalPrice,
        uint256 indexed debtRatio
    );
    event ControlVariableAdjustment(
        uint256 initialBCV,
        uint256 newBCV,
        uint256 adjustment,
        bool addition
    );

    /* ======== STRUCTS ======== */

    // Info for bond holder
    struct Bond {
        uint256 payout; // earthFundToken remaining to be paid
        uint256 vesting; // Blocks left to vest
        uint256 lastBlock; // Last interaction
        uint256 pricePaid; // In DAI, for front end viewing
    }

    // Info for specific bond pairs
    struct BondPair {
        string name; // Currency that user is paying with
        bool active; // Bool to see if bonds are accepting deposits
        uint256 currentPrice; // Current price of the bond
        uint256 decimals; // Number of decimals to get to the smallest denomination of the base curre
        uint256 lastDecay; // Last block that pair was purchased
        uint256 incrementRate; // Rate to increment in smallest denomination (wei for ETH, etc.)
        uint256 decrementRate; // Rate to increment in smallest denomination (wei for ETH, etc.)
        uint256 vestingTerm; // in blocks
        uint256 maxPayout; // in thousandths of a %. i.e. 500 = 0.5%
        uint256 buffer; // How long to wait before adjusting the price, in blocks
    }

    /* ======== STATE VARIABLES ======== */

    address public immutable earthFundToken; // token given as payment for bond
    address payable public immutable treasury; // EarthFund DAO Treasury address

    address public staking; // to auto-stake payout
    address public stakingHelper; // to stake and claim if no staking warmup
    bool public useHelper;

    BondPair[] public bondPairs; // stores bond pairs, as well as their adjustment variables

    mapping(address => mapping(uint256 => Bond)) public bondInfo; // stores bond information for depo
    mapping(string => address) public erc20TokensAddresses; // stores ERC20 token addresses for addit

    constructor(
        address _earthFundToken, // token given as payment for bond
        address payable _treasury // receives profit share from bond
    ) {
        require(_earthFundToken != address(0));
        earthFundToken = _earthFundToken;

        require(_treasury != address(0));
        treasury = _treasury;
    }

    /**
     *  @notice Initialize a new Bond pair that users can purchase
     *  @param _bondName string - The name of the new bond pair
     *  @param _startingPrice uint
     *  @param _vestingTerm uint
     */
    function initializeNewBondTerm(
        string calldata _bondName,
        uint256 _startingPrice,
        uint256 _decimals,
        uint256 _vestingTerm,
        uint256 _incrementRate,
        uint256 _decrementRate,
        uint256 _buffer,
        uint256 _maxPayout,
        bool _isErc20,
        address _tokenAddress
    ) external onlyOwner {
        // We want to increase the price faster than it decays
        require(
            _incrementRate > _decrementRate,
            "Bonding: Increment rate should be higher than decrement rate"
        );

        bondPairs.push(
            BondPair({
                name: _bondName,
                active: true,
                currentPrice: _startingPrice,
                decimals: _decimals,
                lastDecay: block.number,
                buffer: _buffer,
                incrementRate: _incrementRate,
                decrementRate: _decrementRate,
                vestingTerm: _vestingTerm,
                maxPayout: _maxPayout
            })
        );

        if(_isErc20) {
            erc20TokensAddresses[_bondName] = _tokenAddress;
        }
    }

    function bondPrice(uint256 _bondIndex) public view returns (uint256) {
        BondPair memory bond = bondPairs[_bondIndex];
        uint256 timeElapsedInBlocks = block.number - bond.lastDecay;
        uint256 timeElapsed = timeElapsedInBlocks.div(bond.buffer);
        uint256 discount = bond.decrementRate * timeElapsed;
        uint256 discountedPrice = bond.currentPrice - discount;

        if (discountedPrice < 100) {
            discountedPrice = 100;
        }

        return discountedPrice;
    }

    /**
     *  @notice Determine max bond size
     *  @return uint
     */
    function maxPayout(uint256 _bondPairIndex) public view returns (uint256) {
        BondPair memory bond = bondPairs[_bondPairIndex];

        // Remember, the "maxPayout" variable on terms is a percentage of the total
        // supply that can be bonded
        return
            IERC20(earthFundToken)
                .balanceOf(address(this))
                .mul(bond.maxPayout)
                .div(1e5);
    }

    /**
```

```
     *  @notice Deposit bond
     *  @param _bondPairIndex uint      The bond index
     *  @param _amount        uint      Amount of 1EARTH that the user wants to buy
     *  @param _depositor     address   The address that the 1EARTH will go to
     *  @return uint
     */
    function deposit(
        uint256 _bondPairIndex,
        uint256 _amount,
        address _depositor
    ) public payable returns (uint256) {
        require(_depositor != address(0), "Invalid address");
        require(
            _amount < IERC20(earthFundToken).balanceOf(address(this)),
            "Bonding: Not enough in contract to support bond"
        );

        BondPair memory currentBondPair = bondPairs[_bondPairIndex];

        uint256 currentBondPrice = bondPrice(_bondPairIndex);
        uint256 payout = payoutFor(_bondPairIndex, _amount); // payout to bonder is computed

        // require(payout >= 5e18, "Bond too small"); // must be >= 5 1EARTH
        require(payout <= maxPayout(_bondPairIndex), "Bond too large"); // size protection because th

        // If the bonding index is 0, then it's ETH. If not, then it is a token pair.
        if (_bondPairIndex == 0) {
            require(
                msg.value >=
                    _amount.div(10**currentBondPair.decimals).mul(
                        currentBondPrice
                    ),
                "Bonding: Not enough sent for bond"
            );

            treasury.transfer(msg.value);
        } else {
            address tokenAddress = erc20TokensAddresses[currentBondPair.name];
            IERC20(tokenAddress).safeTransferFrom(
                msg.sender,
                treasury,
                _amount
            );
        }

        // depositor info is stored
        bondInfo[_depositor][_bondPairIndex] = Bond({
            payout: bondInfo[_depositor][_bondPairIndex].payout.add(payout),
            vesting: currentBondPair.vestingTerm,
            lastBlock: block.number,
            pricePaid: currentBondPrice
        });

        emit BondCreated(
            _amount,
            payout,
            block.number.add(currentBondPair.vestingTerm),
            currentBondPair.currentPrice
        );

        // Adjust bond decay rate and increase price since this is a purchase
        currentBondPair.lastDecay = block.number;
        currentBondPair.currentPrice = currentBondPair.currentPrice.add(
            currentBondPair.incrementRate
        );
        return payout;
    }

    /**
     *  @notice redeem bond for user
     *  @param _recipient address
     *  @return uint
     */
    function redeem(address _recipient, uint256 _bondPairIndex)
        external
        returns (bool)
    {
        Bond memory userBond = bondInfo[_recipient][_bondPairIndex];
        uint256 percentVested = percentVestedFor(_recipient, _bondPairIndex); // (blocks since last i

        if (percentVested >= 10000) {
            // if fully vested
            delete bondInfo[_recipient][_bondPairIndex]; // delete user info
            emit BondRedeemed(_recipient, userBond.payout, 0); // emit bond data
            return IERC20(earthFundToken).transfer(_recipient, userBond.payout); // send payout
        } else {
            // if unfinished
            // calculate payout vested
            uint256 payout = userBond.payout.mul(percentVested).div(10000);

            // store updated deposit info
            bondInfo[_recipient][_bondPairIndex] = Bond({
                payout: userBond.payout.sub(payout),
                vesting: userBond.vesting.sub(
                    block.number.sub(userBond.lastBlock)
                ),
                lastBlock: block.number,
                pricePaid: userBond.pricePaid
            });

            emit BondRedeemed(
                _recipient,
                payout,
                bondInfo[_recipient][_bondPairIndex].payout
            );
            return IERC20(earthFundToken).transfer(_recipient, payout); // send payout
        }
    }

    enum PARAMETER {
        VESTING,
        MAXPAYOUT,
        INCREMENT,
        DECREMENT,
        BUFFER
    }

    /**
     *  @notice Set or update specific parameter for a specific bond
     *  @param _bondPairIndex Which bond to update, as referenced in bondPairs variable
     *  @param _parameter     What parameter in a bond's terms to update
     *  @param _input         The value for the new term value
     */
    function setBondTerms(
        uint256 _bondPairIndex,
        PARAMETER _parameter,
        uint256 _input
    ) external view onlyOwner {
        BondPair memory bond = bondPairs[_bondPairIndex];

        if (_parameter == PARAMETER.VESTING) {
            // 0
            require(_input >= 86400, "Vesting must be longer than 24 hours");
            bond.vestingTerm = _input;
        } else if (_parameter == PARAMETER.MAXPAYOUT) {
            // 1
            require(_input <= 5000, "Payout cannot be above 5 percent");
            bond.maxPayout = _input;
        } else if (_parameter == PARAMETER.INCREMENT) {
            // 2
            bond.incrementRate = _input;
        } else if (_parameter == PARAMETER.DECREMENT) {
            // 3
            bond.decrementRate = _input;
        } else if (_parameter == PARAMETER.BUFFER) {
            // 4
            bond.buffer = _input;
        }
    }

    /* ======== INTERNAL HELPER FUNCTIONS ======== */

    // VIEW FUNCTIONS //

    /**
     *  @notice calculate how far into vesting a depositor is
     *  @param _depositor address
     *  @return percentVested_ uint
     */
    function percentVestedFor(address _depositor, uint256 _bondPairIndex)
        public
        view
        returns (uint256 percentVested_)
    {
        Bond memory bond = bondInfo[_depositor][_bondPairIndex];
        uint256 blocksSinceLast = block.number.sub(bond.lastBlock);
        uint256 vesting = bond.vesting;

        if (vesting > 0) {
            percentVested_ = blocksSinceLast.mul(10000).div(vesting);
        } else {
            percentVested_ = 0;
        }
    }

    /**
     *  @notice calculate amount of 1EARTH available for claim by depositor
     *  @param _depositor address
     *  @return pendingPayout_ uint
     */
    function pendingPayoutFor(address _depositor, uint256 _bondPairIndex)
        external
        view
        returns (uint256 pendingPayout_)
    {
        uint256 percentVested = percentVestedFor(_depositor, _bondPairIndex);
        uint256 payout = bondInfo[_depositor][_bondPairIndex].payout;

        if (percentVested >= 10000) {
            pendingPayout_ = payout;
        } else {
            pendingPayout_ = payout.mul(percentVested).div(10000);
        }
    }

    /**
     *  @notice calculate interest due for new bond
     *  @param _value uint
     *  @return uint
     */
    function payoutFor(uint256 _bondPairIndex, uint256 _value)
        public
        view
        returns (uint256)
    {
        uint256 price = bondPrice(_bondPairIndex);
        return _value.div(price);
    }
}
```

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 specifiers 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