
Armors Labs
DEUS Lending
Smart Contract Audit

DEUS Lending Audit Summary

Project name : DEUS Lending Audit Contract

Project address: None

Code URL : https://github.com/deusfinance/lending-audit

Commit : 16ec7ae6e8bf082e89131926e8e988aca4cdce9c

Project target : DEUS Lending Contract Audit

Blockchain : Fantom

Test result : PASSED

Audit Info

Audit NO : 0X202203270006

Audit Team : Armors Labs

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

DEUS Lending Audit

The DEUS Lending team asked us to review and audit their DEUS Lending 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
DEUS Lending Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2022-03-27

Audit results

Warning:

1. Owner has permission to retrieve the Solidex Holder ERC20 Token "in case of emergency"
2. The project relies on third party contract interfaces and this report does not include the security implications of third party contracts

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the DEUS Lending 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
./Oracle.sol	ed45d9b6676df9e8c53bf78d0346fae3
./SolidexHolder.sol	97a3ac519a4c8860b402c070079632f1
./DeiLenderSolidex.sol	8421a36e8638600e189daf1b254e07a7
./MintHelper.sol	f6c559e7f56e65adf6ba220a287e8f26
./interfaces/IDEIStablecoin.sol	4ce9d803c93e21fd9c02007254535c86
./interfaces/IMintHelper.sol	756af2baa5d28030ae4f366a891528fb

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

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

Contract file

```
// Be name Khoda
// Bime Abolfazl
// SPDX-License-Identifier: GPL3.0-or-later

// ===================================================================================================
//  _|_|_|    _|_|_|_|  _|    _|  _|_|_|      _|_|_|_|  _|
//  _|    _|  _|        _|    _|  _|            _|            _|_|_|      _|_|_|  _|_|_|      _|_|_|
//  _|    _|  _|_|_|    _|    _|    _|_|        _|_|_|    _|  _|    _|  _|  _|    _|  _|    _|  _|
//  _|    _|  _|        _|    _|        _|      _|        _|  _|    _|  _|  _|    _|  _|    _|  _|
//  _|_|_|    _|_|_|_|    _|_|    _|_|_|        _|        _|  _|    _|    _|_|_|  _|    _|    _|_|_|
// ===================================================================================================
// ================== DEI Lender Solidex ==================
// ========================================================
// DEUS Finance: https://github.com/deusfinance

// Primary Author(s)
// MRM: https://github.com/smrm-dev
// MMD: https://github.com/mmd-mostafaee

// Reviewer(s)
// Vahid: https://github.com/vahid-dev
// HHZ: https://github.com/hedzed

pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;
import "./interfaces/IMintHelper.sol";
import "@boringcrypto/boring-solidity/contracts/libraries/BoringMath.sol";
import "@boringcrypto/boring-solidity/contracts/BoringOwnable.sol";
import "@boringcrypto/boring-solidity/contracts/ERC20.sol";
```

```
import "@boringcrypto/boring-solidity/contracts/interfaces/IERC20.sol";
import "@boringcrypto/boring-solidity/contracts/interfaces/IMasterContract.sol";
import "@boringcrypto/boring-solidity/contracts/libraries/BoringRebase.sol";
import "@boringcrypto/boring-solidity/contracts/libraries/BoringERC20.sol";
import {SolidexHolder as Holder} from "./SolidexHolder.sol";

interface LpDepositor {
    function getReward(address[] calldata pools) external;
}

interface HIERC20 {
    function transfer(address recipient, uint256 amount)
        external
        returns (bool);
}

interface IOracle {
    function getPrice() external view returns (uint256);
}

contract DeiLenderSolidex is BoringOwnable {
    using BoringMath for uint256;
    using BoringMath128 for uint128;
    using RebaseLibrary for Rebase;
    using BoringERC20 for IERC20;

    event UpdateAccrue(uint256 interest);
    event Borrow(address from, address to, uint256 amount, uint256 debt);
    event Repay(address from, address to, uint256 amount, uint256 repayAmount);
    event AddCollateral(address from, address to, uint256 amount);
    event RemoveCollateral(address from, address to, uint256 amount);

    IERC20 public collateral;

    IERC20 public solid;
    IERC20 public solidex;
    address public lpDepositor;
    uint256 public maxCap;

    IOracle public oracle;

    uint256 public BORROW_OPENING_FEE;

    uint256 public LIQUIDATION_RATIO;

    uint256 public totalCollateral;
    Rebase public totalBorrow;

    mapping(address => uint256) public userCollateral;
    mapping(address => uint256) public userBorrow;
    mapping(address => address) public userHolder;

    address public mintHelper;

    struct AccrueInfo {
        uint256 lastAccrued;
        uint256 feesEarned;
        uint256 interestPerSecond;
    }

    AccrueInfo public accrueInfo;

    constructor(
        IERC20 collateral_,
        IOracle oracle_,
        IERC20 solid_,
```

```
        IERC20 solidex_,
        address lpDepositor_,
        uint256 maxCap_,
        uint256 interestPerSecond_,
        uint256 borrowOpeningFee,
        uint256 liquidationRatio,
        address mintHelper_
    ) public {
        collateral = collateral_;
        accrueInfo.interestPerSecond = interestPerSecond_;
        accrueInfo.lastAccrued = block.timestamp;
        BORROW_OPENING_FEE = borrowOpeningFee;
        LIQUIDATION_RATIO = liquidationRatio;
        oracle = oracle_;
        solid = solid_;
        solidex = solidex_;
        lpDepositor = lpDepositor_;
        maxCap = maxCap_;
        mintHelper = mintHelper_;
    }

    function setOracle(IOracle oracle_) external onlyOwner {
        oracle = oracle_;
    }

    function setMaxCap(uint256 maxCap_) external onlyOwner {
        maxCap = maxCap_;
    }

    function setBorrowOpeningFee(uint256 borrowOpeningFee_) external onlyOwner {
        BORROW_OPENING_FEE = borrowOpeningFee_;
    }

    function setLiquidationRatio(uint256 liquidationRatio_) external onlyOwner {
        LIQUIDATION_RATIO = liquidationRatio_;
    }

    function setMintHelper(address mintHelper_) external onlyOwner {
        mintHelper = mintHelper_;
    }

    function getRepayAmount(uint256 amount)
        public
        view
        returns (uint256 repayAmount)
    {
        Rebase memory _totalBorrow = totalBorrow;
        (uint128 elastic, ) = getCurrentElastic();
        _totalBorrow.elastic = elastic;
        (_totalBorrow, repayAmount) = _totalBorrow.sub(amount, true);
    }

    /// returns user total debt (borrowed amount + interest)
    function getDebt(address user) public view returns (uint256 debt) {
        if (totalBorrow.base == 0) return 0;

        (uint128 elastic, ) = getCurrentElastic();
        return userBorrow[user].mul(uint256(elastic)) / totalBorrow.base;
    }

    /// returns liquidation price for requested user
    function getLiquidationPrice(address user) public view returns (uint256) {
        uint256 userCollateralAmount = userCollateral[user];
        if (userCollateralAmount == 0) return 0;

        uint256 liquidationPrice = (getDebt(user).mul(1e18).mul(1e18)) /
```

```
            (userCollateralAmount.mul(LIQUIDATION_RATIO));
        return liquidationPrice;
    }

    /// returns withdrawable amount for requested user
    function getWithdrawableCollateralAmount(address user)
        public
        view
        returns (uint256)
    {
        uint256 userCollateralAmount = userCollateral[user];
        if (userCollateralAmount == 0) return 0;

        uint256 neededCollateral = (getDebt(user).mul(1e18).mul(1e18)) /
            (oracle.getPrice().mul(LIQUIDATION_RATIO));

        return
            userCollateralAmount > neededCollateral
                ? userCollateralAmount - neededCollateral
                : 0;
    }

    function isSolvent(address user) public view returns (bool) {
        // accrue must have already been called!

        uint256 userCollateralAmount = userCollateral[user];
        if (userCollateralAmount == 0) return getDebt(user) == 0;

        return
            userCollateralAmount.mul(oracle.getPrice()).mul(LIQUIDATION_RATIO) /
                (uint256(1e18).mul(1e18)) >
            getDebt(user);
    }

    function getCurrentElastic()
        internal
        view
        returns (uint128 elastic, uint128 interest)
    {
        Rebase memory _totalBorrow = totalBorrow;
        uint256 elapsedTime = block.timestamp - accrueInfo.lastAccrued;
        if (elapsedTime != 0 && _totalBorrow.base != 0) {
            interest = (uint256(_totalBorrow.elastic)
                .mul(accrueInfo.interestPerSecond)
                .mul(elapsedTime) / 1e18).to128();
            elastic = _totalBorrow.elastic.add(interest);
        } else {
            return (totalBorrow.elastic, 0);
        }
    }

    function accrue() public {
        uint256 elapsedTime = block.timestamp - accrueInfo.lastAccrued;
        if (elapsedTime == 0) return;
        if (totalBorrow.base == 0) {
            accrueInfo.lastAccrued = uint256(block.timestamp);
            return;
        }

        (uint128 elastic, uint128 interest) = getCurrentElastic();

        accrueInfo.lastAccrued = uint256(block.timestamp);
        totalBorrow.elastic = elastic;
        accrueInfo.feesEarned = accrueInfo.feesEarned.add(interest);

        emit UpdateAccrue(interest);
```

```
    }

    function addCollateral(address to, uint256 amount) public {
        userCollateral[to] = userCollateral[to].add(amount);
        totalCollateral = totalCollateral.add(amount);
        if (userHolder[to] == address(0)) {
            Holder holder = new Holder(lpDepositor, address(this), to);
            userHolder[to] = address(holder);
        }
        collateral.safeTransferFrom(msg.sender, userHolder[to], amount);
        emit AddCollateral(msg.sender, to, amount);
    }

    function removeCollateral(address to, uint256 amount) public {
        accrue();
        userCollateral[msg.sender] = userCollateral[msg.sender].sub(amount);

        totalCollateral = totalCollateral.sub(amount);

        Holder(userHolder[msg.sender]).withdrawERC20(
            address(collateral),
            to,
            amount
        );

        require(isSolvent(msg.sender), "User is not solvent!");
        emit RemoveCollateral(msg.sender, to, amount);
    }

    function borrow(address to, uint256 amount) public returns (uint256 debt) {
        accrue();
        uint256 fee = amount.mul(BORROW_OPENING_FEE) / 1e18;
        (totalBorrow, debt) = totalBorrow.add(amount.add(fee), true);
        accrueInfo.feesEarned = accrueInfo.feesEarned.add(fee);
        userBorrow[msg.sender] = userBorrow[msg.sender].add(debt);

        require(
            totalBorrow.elastic <= maxCap,
            "Lender total borrow exceeds cap"
        );
        require(isSolvent(msg.sender), "User is not solvent!");
        IMintHelper(mintHelper).mint(to, amount);
        emit Borrow(msg.sender, to, amount.add(fee), debt);
    }

    function repayElastic(address to, uint256 debt)
        public
        returns (uint256 repayAmount)
    {
        accrue();

        uint256 amount = debt.mul(totalBorrow.base) / totalBorrow.elastic;

        (totalBorrow, repayAmount) = totalBorrow.sub(amount, true);
        userBorrow[to] = userBorrow[to].sub(amount);

        IMintHelper(mintHelper).burnFrom(msg.sender, repayAmount);

        emit Repay(msg.sender, to, amount, repayAmount);
    }

    function repayBase(address to, uint256 amount)
        public
        returns (uint256 repayAmount)
    {
        accrue();
```

```
        (totalBorrow, repayAmount) = totalBorrow.sub(amount, true);
        userBorrow[to] = userBorrow[to].sub(amount);

        IMintHelper(mintHelper).burnFrom(msg.sender, repayAmount);

        emit Repay(msg.sender, to, amount, repayAmount);
    }

    function liquidate(address[] calldata users, address to) public {
        accrue();

        uint256 totalCollateralAmount;
        uint256 totalDeiAmount;

        for (uint256 i = 0; i < users.length; i++) {
            address user = users[i];

            if (!isSolvent(user)) {
                uint256 amount = userBorrow[user];

                uint256 deiAmount;
                (totalBorrow, deiAmount) = totalBorrow.sub(amount, true);

                totalDeiAmount += deiAmount;
                totalCollateralAmount += userCollateral[user];

                emit RemoveCollateral(user, to, userCollateral[user]);
                emit Repay(msg.sender, user, amount, deiAmount);

                Holder(userHolder[user]).withdrawERC20(
                    address(collateral),
                    to,
                    userCollateral[user]
                );
                userCollateral[user] = 0;
                userBorrow[user] = 0;
            }
        }

        require(totalDeiAmount != 0, "All users are solvent");

        IMintHelper(mintHelper).burnFrom(msg.sender, totalDeiAmount);
    }

    function withdrawFees(address to, uint256 amount) public onlyOwner {
        accrue();

        IMintHelper(mintHelper).mint(to, amount);
        accrueInfo.feesEarned = accrueInfo.feesEarned.sub(amount);
    }

    function claim(address[] calldata pools) public {
        Holder(userHolder[msg.sender]).claim(pools);
    }

    function claimAndWithdraw(address[] calldata pools, address to) public {
        Holder(userHolder[msg.sender]).claim(pools);
        Holder(userHolder[msg.sender]).withdrawERC20(
            address(solid),
            to,
            solid.balanceOf(userHolder[msg.sender])
        );
        Holder(userHolder[msg.sender]).withdrawERC20(
            address(solidex),
            to,
```

```
            solidex.balanceOf(userHolder[msg.sender])
        );
    }

    function emergencyHolderWithdraw(
        address holder,
        address token,
        address to,
        uint256 amount
    ) public onlyOwner {
        Holder(holder).withdrawERC20(token, to, amount);
    }

    function emergencyWithdraw(
        address token,
        address to,
        uint256 amount
    ) public onlyOwner {
        HIERC20(token).transfer(to, amount);
    }
}

// Be name Khoda
// Bime Abolfazl
// SPDX-License-Identifier: GPL3.0-or-later

// ========================================================================================================
//  _|_|_|    _|_|_|_|  _|    _|    _|_|_|      _|_|_|_|  _|                                  _|_|_|      _|_|_|
//  _|    _|  _|        _|    _|  _|            _|        _|_|_|      _|_|_|  _|_|_|      _|_|_|
//  _|    _|  _|_|_|    _|    _|    _|_|        _|_|_|    _|    _|  _|    _|  _|    _|  _|    _|
//  _|    _|  _|        _|    _|        _|      _|        _|    _|  _|    _|  _|    _|  _|    _|
//  _|_|_|    _|_|_|_|    _|_|    _|_|_|        _|        _|    _|    _|_|_|  _|    _|    _|_|_|
// ========================================================================================================
// ==================== Mint Helper ===================
// ====================================================
// DEUS Finance: https://github.com/deusfinance

// Primary Author(s)
// Vahid: https://github.com/vahid-dev

pragma solidity 0.8.12;

import "@openzeppelin/contracts/access/AccessControl.sol";
import "./interfaces/IMintHelper.sol";
import "./interfaces/IDEIStablecoin.sol";

/// @title Mint Helper
/// @author DEUS Finance
/// @notice DEI minter contract for lending contracts
contract MintHelper is IMintHelper, AccessControl {
    address public dei;
    mapping(address => bool) public useVirtualReserve;
    uint256 public virtualReserve;
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");

    modifier isMinter() {
        require(hasRole(MINTER_ROLE, msg.sender), "Caller is not minter");
        _;
    }

    modifier isAdmin() {
        require(hasRole(DEFAULT_ADMIN_ROLE, msg.sender), "Caller is not admin");
        _;
    }

    constructor(
```

```
        address dei_,
        uint256 virtualReserve_,
        address admin
    ) {
        dei = dei_;
        virtualReserve = virtualReserve_;
        _setupRole(DEFAULT_ADMIN_ROLE, admin);
    }

    /// @notice mint DEI for user
    /// @param recv DEI reciever
    /// @param amount DEI amount
    function mint(address recv, uint256 amount) external isMinter {
        if (useVirtualReserve[msg.sender]) {
            virtualReserve += amount;
        }
        IDEIStablecoin(dei).pool_mint(recv, amount);
    }

    /// @notice burn DEI from user
    /// @param from burnt user
    /// @param amount DEI amount
    function burnFrom(address from, uint256 amount) external isMinter {
        if (useVirtualReserve[msg.sender]) {
            virtualReserve -= amount;
        }
        IDEIStablecoin(dei).burnFrom(from, amount);
    }

    /// @notice This function use pool feature to manage buyback and recollateralize on DEI minter po
    /// @dev simulates the collateral in the contract
    /// @param collat_usd_price pool's collateral price (is 1e6) (decimal is 6)
    /// @return amount of collateral in the contract
    function collatDollarBalance(uint256 collat_usd_price)
        public
        view
        returns (uint256)
    {
        uint256 deiCollateralRatio = IDEIStablecoin(dei)
            .global_collateral_ratio();
        return (virtualReserve * collat_usd_price * deiCollateralRatio) / 1e12;
    }

    /// @notice sets virtualReserve
    /// @dev only admin can call function
    /// @param virtualReserve_ new virtualReserve amount
    function setVirtualReserve(uint256 virtualReserve_) external isAdmin {
        virtualReserve = virtualReserve_;
    }

    /// @notice set useVirtualReserve for specific minter
    /// @dev only admin can call function
    function setUseVirtualReserve(address pool, bool state) external isAdmin {
        useVirtualReserve[pool] = state;
    }
}

// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.12;

import "@openzeppelin/contracts/access/AccessControl.sol";

interface IERC20 {
    function balanceOf(address account) external view returns (uint256);
```

```
    function totalSupply() external view returns (uint256);
}

struct SchnorrSign {
    uint256 signature;
    address owner;
    address nonce;
}

interface IMuonV02 {
    function verify(
        bytes calldata reqId,
        uint256 hash,
        SchnorrSign[] calldata _sigs
    ) external returns (bool);
}

contract Oracle is AccessControl {
    uint256 public validTime;
    uint256 public twapPrice;
    uint256 public minReqSigs;
    uint256 public lastTimestamp;
    address public lp;
    address public muonContract;
    uint32 public APP_ID;

    modifier isAdmin() {
        require(
            hasRole(DEFAULT_ADMIN_ROLE, msg.sender),
            "ORACLE: caller is not admin"
        );
        _;
    }

    constructor(
        address lp_,
        address muon_,
        address admin,
        uint256 minReqSigs_,
        uint256 validTime_,
        uint256 twapPrice_,
        uint32 appId
    ) {
        lp = lp_;
        muonContract = muon_;
        minReqSigs = minReqSigs_;
        validTime = validTime_;
        twapPrice = twapPrice_;
        lastTimestamp = block.timestamp;
        APP_ID = appId;
        _setupRole(DEFAULT_ADMIN_ROLE, admin);
    }

    function updatePrice(
        uint256 twapPrice_,
        uint256 timestamp,
        bytes calldata _reqId,
        SchnorrSign[] calldata sigs
    ) external {
        require(
            sigs.length >= minReqSigs,
            "ORACLE: insufficient number of signatures"
        );
        require(timestamp > lastTimestamp, "ORACLE: price is expired");
        bytes32 hash = keccak256(
            abi.encodePacked(APP_ID, lp, twapPrice_, timestamp)
```

```
        );
        require(
            IMuonV02(muonContract).verify(_reqId, uint256(hash), sigs),
            "ORACLE: not verified"
        );
        lastTimestamp = block.timestamp;
        twapPrice = twapPrice_;
    }

    function getPrice() external view returns (uint256) {
        require(
            (block.timestamp - lastTimestamp) <= validTime,
            "ORACLE: price is not valid"
        );
        return twapPrice;
    }

    function setMuonContract(address muon) external isAdmin {
        muonContract = muon;
    }

    function setMinReqSig(uint256 minReqSigs_) external isAdmin {
        minReqSigs = minReqSigs_;
    }

    function setValidTime(uint256 validTime_) external isAdmin {
        validTime = validTime_;
    }

    function setAppId(uint8 APP_ID_) external isAdmin {
        APP_ID = APP_ID_;
    }
}

// Be name Khoda
// Bime Abolfazl
// SPDX-License-Identifier: MIT

// ====================================================================================================
//  _|_|_|    _|_|_|_|  _|      _|  _|_|_|      _|_|_|_|  _|
//  _|    _|  _|        _|_|    _|  _|    _|        _|            _|_|_|      _|_|_|  _|_|_|      _|_|_|
//  _|    _|  _|_|_|    _|  _|  _|  _|_|_|        _|_|_|    _|  _|    _|  _|    _|  _|    _|  _|  _|
//  _|    _|  _|        _|    _|_|  _|    _|      _|        _|  _|    _|  _|    _|  _|    _|  _|  _|
//  _|_|_|    _|_|_|_|  _|      _|  _|_|_|        _|        _|  _|    _|    _|_|_|  _|    _|    _|_|_|
// ====================================================================================================
// ========== Holder ==========
// ============================================
// DEUS Finance: https://github.com/deusfinance

// Primary Author(s)
// Mmd: https://github.com/mmd-motafaee

pragma solidity 0.6.12;

interface LpDepositor {
    function getReward(address[] calldata pools) external;
}

interface HIERC20 {
    function transfer(address recipient, uint256 amount)
        external
        returns (bool);
}

contract SolidexHolder {
    LpDepositor public lpDepositor;
```

```
    address public lender;
    address public user;

    constructor(
        address lpDepositor_,
        address lender_,
        address user_
    ) public {
        lpDepositor = LpDepositor(lpDepositor_);
        lender = lender_;
        user = user_;
    }

    function claim(address[] calldata pools) public {
        lpDepositor.getReward(pools);
    }

    function withdrawERC20(
        address token,
        address to,
        uint256 amount
    ) public returns (bool) {
        require(msg.sender == lender, "SolidexHolder: You are not lender");
        HIERC20(token).transfer(to, amount);
        return true;
    }
}

//Dar panah khoda
```


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