Armors Labs

Voyager Pass

Smart Contract Audit

Voyager Pass Audit Summary

Project name : Voyager Pass Contract

Project address: None

Code URL : https://etherscan.io/address/0xCcA0ab80494824Fb3E1aCce8d02BcA3C1dB32165#code

Commit : None

Project target : Voyager Pass Contract Audit

Blockchain : Ethereum

Test result : PASSED

Audit Info

Audit NO : 0X202205180016

Audit Team : Armors Labs

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

Voyager Pass Audit

The Voyager Pass team asked us to review and audit their Voyager Pass 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
Voyager Pass Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2022-05-18

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 Voyager Pass 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
VoyagerPass.sol	6c2d6bbe568c6237dfd99994bbfa6ba7

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

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.4;

import "@openzeppelin/contracts/utils/introspection/IERC165.sol";
import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
import "@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";
import "@openzeppelin/contracts/token/ERC721/extensions/IERC721Metadata.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/utils/Context.sol";

import "@openzeppelin/contracts/utils/introspection/ERC165.sol";
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/token/ERC721/extensions/IERC721Enumerable.sol";
import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
import "@openzeppelin/contracts/utils/Strings.sol";
import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";


library MintUtil {
    using SafeMath for uint256;
    using ECDSA for bytes32;
    using Strings for uint256;

    function getDiscountedPrice(uint _startTime, uint256 _startPrice, uint256 _discountRate, uint256
    uint256 _duration) public view returns (uint256) {
        uint256 timeElapsed = block.timestamp.sub(_startTime);
        if(timeElapsed >= _duration){
            return _endPrice;
        }
        uint256 discount = timeElapsed.mul(_discountRate);
        uint256 discountedPrice = _startPrice.sub(discount);
        return discountedPrice.div(100000000000000).mul(100000000000000);
    }

    function inWhitelist(bytes32 _leaf, bytes32 _merkelRoot, bytes32[] calldata _merkleProof) public
        return MerkleProof.verify(_merkleProof, _merkelRoot, _leaf);
    }

    function canMint(uint _whitelistIndex, bool _voyagerCaptainMintActive,
    bool _legacyMintActive, bool _privateMintActive, bool _publicSaleActive,
    bool _addressMinted) public pure returns (bool, string memory) {
```

```
        if (!((_whitelistIndex == 0 && _voyagerCaptainMintActive) ||
        (_whitelistIndex == 1 && _legacyMintActive) ||
        (_whitelistIndex == 2 && _privateMintActive) ||
        (_whitelistIndex == 100 && _publicSaleActive))){
            return (false, "Your address is not yet qualified to mint, please read the FAQ and check
        }

        if (_addressMinted) {
            return (false, "You can only mint one per wallet");
        }

        return (true, "");
    }

    function canMint2(uint _whitelistIndex, uint256 _voyagerCaptainMintedAmount, uint256 _voyagerCapt
    uint256 _privateMintLimit, uint256 _publicSaleMintedAmount, uint256 _publicSaleMintLimit) public
        if(_whitelistIndex == 0 && _voyagerCaptainMintedAmount.add(1) > _voyagerCaptainMintLimit){
            return(false,  "Can't mint more than voyager captain limit");
        } else if(_whitelistIndex == 1 && _legacyMintedAmount.add(1) > _legacyMintLimit){
            return(false,   "Can't mint more than legacy mint limit");
        } else if(_whitelistIndex == 2 && _privateMintedAmount.add(1) > _privateMintLimit){
            return(false, "Can't mint more than private mint limit");
        } else if(_whitelistIndex == 100 && _publicSaleMintedAmount.add(1) > _publicSaleMintLimit){
            return(false, "Can't mint more than public sale limit");
        }
        return (true, "");

    }

    function getTokenURI(uint256 _tokenId, bool _reveal, string memory _blindURI, string memory _base
        if (!_reveal) {
            return string(abi.encodePacked(_blindURI, _tokenId.toString()));
        } else {
            return string(abi.encodePacked(_baseURI, _tokenId.toString()));
        }
    }
}


contract VoyagerPass is ERC721("Voyager Pass", "VOP"), ERC721Enumerable, Ownable, ReentrancyGuard {
    using SafeMath for uint256;

    string private baseURI;
    string private blindURI;
    uint256 private constant TOTAL_NFT = 10000;
    // initial price for the auction
    uint256 public mintPrice = 1 ether;
    uint256 public endingPrice = 0.15 ether;
    uint256 public privatePrice = 0.1 ether;
    uint256 public auctionDuration = 5400 seconds;
    // 1.5 hours = 5400 seconds, so price decrease (1 - 0.1) * 10**18 / 5400 = 166_666_666_666_666 we
    uint256 public discountRate = 166_666_666_666_666 wei;

    bool public reveal;

    bool public voyagerCaptainMintActive;
    bool public legacyMintActive;
    bool public privateMintActive;
    bool public publicSaleActive;
    bool public dutchAuctionActive;

    // save all three roots
    bytes32[3] public whitelistInfo;

    mapping (address => bool) public addressMinted;
    mapping (uint256 => bool) public isPaid;
```

```
uint256 public voyagerCaptainMintLimit = 500;
uint256 public legacyMintLimit = 1200;
uint256 public privateMintLimit = 8300;
uint256 public publicSaleMintLimit = 0;
uint256 public voyagerCaptainMintedAmount;
uint256 public legacyMintedAmount;
uint256 public privateMintedAmount;
uint256 public publicSaleMintedAmount;
uint256 public dutchAuctionStartAt;

function revealNow() external onlyOwner {
    reveal = true;
}

function setMintActive(bool _isActive, uint mintTypeIndex) external onlyOwner {
    if (mintTypeIndex == 0)
        voyagerCaptainMintActive = _isActive;
    else if (mintTypeIndex == 1)
        legacyMintActive = _isActive;
    else if (mintTypeIndex == 2)
        privateMintActive = _isActive;
    else if( mintTypeIndex == 3)
        publicSaleActive = _isActive;
}

function setDutchAuctionActive(bool _dutchAuctionActive) external onlyOwner {
    dutchAuctionActive = _dutchAuctionActive;
    dutchAuctionStartAt = block.timestamp;
}

function setPrivatePrice(uint256 _privatePrice) external onlyOwner {
    privatePrice = _privatePrice;
}

function setDutchAuctionInfo(uint256 _startPrice, uint256 _endPrice, uint256 _duration) external
    mintPrice = _startPrice;
    endingPrice = _endPrice;
    auctionDuration = _duration;
    discountRate = (mintPrice.sub(endingPrice)).div(auctionDuration);
}

function setMintLimit(uint256 _voyagerCaptainMintLimit, uint256 _legacyMintLimit, uint256 _privat
    voyagerCaptainMintLimit = _voyagerCaptainMintLimit;
    legacyMintLimit = _legacyMintLimit;
    privateMintLimit = _privateMintLimit;
    publicSaleMintLimit = TOTAL_NFT.sub(voyagerCaptainMintLimit.add(legacyMintLimit).add(privateM
}

function setURIs(string memory _blindURI, string memory _URI) external onlyOwner {
    blindURI = _blindURI;
    baseURI = _URI;
}

function setRoot(bytes32 _root, uint _whitelistIndex) external onlyOwner {
    whitelistInfo[_whitelistIndex] = _root;
}

function airdrop(address _target, uint _whitelistIndex) external onlyOwner {

    require(totalSupply().add(1) <= TOTAL_NFT, "Can't mint more than 10000 NFTs");

    addressMinted[_target] = true;

    uint256 tokenId = totalSupply() + 1;
    updateMintMaps(tokenId, _whitelistIndex);
```

```
        _safeMint(_target,tokenId);
    }

    function canMint(uint _whitelistIndex, address _address) public view returns (bool, string memory
        // check if user is authorized to mint / mint is active / user has already minted
        require(totalSupply().add(1) <= TOTAL_NFT, "Can't mint more than 10000 NFTs");
        return MintUtil.canMint(_whitelistIndex, voyagerCaptainMintActive, legacyMintActive, privateM
        addressMinted[_address]);
    }

    function canMint2(uint _whitelistIndex) public view returns (bool, string memory) {
        // check if minted amount is greater than limit
        return MintUtil.canMint2(_whitelistIndex, voyagerCaptainMintedAmount, voyagerCaptainMintLimit
         legacyMintLimit, privateMintedAmount, privateMintLimit,publicSaleMintedAmount, publicSaleMin
    }


    function getMintPriceByUser(uint _whitelistIndex) public view returns (uint256) {
        if(_whitelistIndex <= 1){
            return 0;
        }else if(_whitelistIndex == 2){
            return privatePrice;
        }else if(dutchAuctionActive){
            return MintUtil.getDiscountedPrice(dutchAuctionStartAt, mintPrice, discountRate, endingPr
        }else{
            return endingPrice;
        }
    }

    function withdraw() public onlyOwner {
        payable(0xfA61b6E35613f014Bd4387898790E89572f63B57).transfer(address(this).balance);
    }

    function updateMintMaps(uint256 _tokenId, uint _whitelistIndex) private {
        // update minted amount for different tiers and the isPaid map
        if(_whitelistIndex == 0){
            voyagerCaptainMintedAmount = voyagerCaptainMintedAmount.add(1);
            isPaid[_tokenId] = false;
        }else if(_whitelistIndex == 1){
            legacyMintedAmount = legacyMintedAmount.add(1);
            isPaid[_tokenId] = false;
        }else if(_whitelistIndex == 2){
            privateMintedAmount = privateMintedAmount.add(1);
            isPaid[_tokenId] = true;
        }else{
            publicSaleMintedAmount = publicSaleMintedAmount.add(1);
            isPaid[_tokenId] = true;
        }
    }

    function getWhitelistIndex(bytes32[] calldata _merkleProof, bytes32 leaf) internal view returns(u
        uint whitelistIndex = 100;
        if(MerkleProof.verify(_merkleProof, whitelistInfo[0], leaf)){
            whitelistIndex = 0;
        } else if(MerkleProof.verify(_merkleProof, whitelistInfo[1], leaf)){
            whitelistIndex = 1;
        } else if(MerkleProof.verify(_merkleProof, whitelistInfo[2], leaf)){
            whitelistIndex = 2;
        }
        return whitelistIndex;
    }

    function mintNFT(bytes32[] calldata _merkleProof) payable external nonReentrant{
        bytes32 leaf = keccak256(abi.encodePacked(msg.sender));
        uint whitelistIndex = getWhitelistIndex(_merkleProof, leaf);
        // apply 2 checks to make sure the mint request is valid
```

```
        (bool success, string memory reason) = canMint(whitelistIndex, msg.sender);
        (bool success2, string memory reason2) = canMint2(whitelistIndex);
        require(success, reason);
        require(success2, reason2);

        uint256 currentPrice = getMintPriceByUser(whitelistIndex);
        require(currentPrice <= msg.value, "Insufficient payable value");

        addressMinted[msg.sender] = true;

        uint256 tokenId = totalSupply() + 1;
        updateMintMaps(tokenId, whitelistIndex);
        _safeMint(msg.sender, tokenId);
        if(dutchAuctionActive){
            uint256 refund = msg.value.sub(currentPrice);
            if (refund > 0) {
                payable(msg.sender).transfer(refund);
            }
        }
    }

    function isTokenPaid(uint256 _tokenId) public view returns (bool) {
        return isPaid[_tokenId];
    }

    function tokenURI(uint256 _tokenId) public view virtual override returns (string memory) {
        require(_exists(_tokenId), "URI query for nonexistent token");
        return MintUtil.getTokenURI(_tokenId, reveal, blindURI, baseURI);
    }

    function supportsInterface(bytes4 _interfaceId) public view override (ERC721, ERC721Enumerable) r
        return super.supportsInterface(_interfaceId);
    }

    function _beforeTokenTransfer(address _from, address _to, uint256 _tokenId) internal override(ERC
        super._beforeTokenTransfer(_from, _to, _tokenId);
    }
}
```


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 vulnerabilities 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