HAECHI AUDIT

MarbleX - NFT Marketplace

Smart Contract Security Analysis

Published on : Oct 21, 2022

Version v1.1

HAECHI AUDIT

Smart Contract Audit Certificate

MarbleX - NFT Marketplace

Security Report Published by HAECHI AUDIT

v1.1 Oct 21, 2022 - final report

v1.0 Oct 14, 2022 - initial report

Auditor : Paul Kim, Allen Roh

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	1	1	-	-
High	1	1	-	-
Medium	1	1	-	-
Low	-	-	-	-
Tips	-	-	-	-

TABLE OF CONTENTS

ABOUT US

The most reliable web3 security partner.

HAECHI AUDIT is a flagship service of HAECHI LABS, the leader of the global blockchain industry. We bring together the best Web2 and Web3 experts. Security Researchers with expertise in cryptography, leaders of the global best hacker team, and blockchain/smart contract experts are responsible for securing your Web3 service.

We have secured the most well-known web3 services including 1inch, SushiSwap, Klaytn, Badger DAO, SuperRare, Netmarble, Klaytn and Chainsafe. We have secured $60B crypto assets on over 400 main-nets, Defi protocols, NFT services, P2E, and Bridges.

HAECHI AUDIT is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

Inquiries: audit@haechi.io

Website: audit.haechi.io

Executive Summary

Purpose of this report

This report was prepared to audit the security of the NFT Marketplace contracts developed by the MarbleX team. HAECHI AUDIT conducted the audit focusing on whether the system created by the MarbleX team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the NFT Marketplace contracts. As the contract is based on Wyvern 2.2 which was widely used, we have focused on the following.

- different codes between the contracts and Wyvern 2.2
- correct fixes of 1-day vulnerabilities of Wyvern 2.2

Codebase Submitted for the Audit

The codes used in this Audit can be found on GitHub

- https://github.com/Bisonai-CIC/klaybay-wyvern-contracts

The commit hash of the code used for this Audit is `7f5b5c0a2dedc52c4f82e164d661b313b0aa56c3`

For the patch review, the last commit hash is `7a5b820d3c7cf1e66a7eab18b1b353bb0243827d`

Audit Timeline

Date	Event
2022/09/22	Audit Initiation
2022/10/14	Delivery of v1.0 report.
2022/10/21	Delivery of v1.1 report.

Findings

HAECHI AUDIT found 1 Critical, 1 High and 1 Medium severity issues.

#ID	Title	Type	Severity	Difficulty
1	Denial of Service can be occured since the contract “AuthenticatedProxy” is not initialized.	Denial of Service	Medium	Low
2	The hashOrder related vulnerability is not patched, leads to potential theft of assets	Cryptography	Critical	Low
3	Logical error in guardedArrayReplace function leads to memory overwriting	Low-Level	High	High

OVERVIEW

Protocol overview

The contract we audited is mostly equal to the contracts of Wyvern v2.2 in this repository.
There are two major components of the project, the NFT Marketplace and the DAO.

The NFT Marketplace works as follows. There's an `Order` structure that contains the information on the order placed. This includes, among other things, maker/taker of the order, exchange address, payment token address, price information, fee information, listing time and expiration time, and the salt for preventing duplicate hashes. The execution of NFT transfers are handled in a much more general way in Wyvern 2.2. It allows the buyer and the seller to decide on a contract call parameters, i.e. the target of the call, the call method (call or delegatecall) and the calldata. To decide on the calldata, both the buyer and the seller decide on their idea on the calldata and the bitmap on which bits can be changed by the other. With some memory manipulation, the contract computes the final calldata based on the two order structures of buyer and seller and executes the contract. To verify that the orders are approved by the buyer or seller, either the appropriate ECDSA signatures should be provided, or they have to call the contract for approvals, or they have to call the contract themselves. In this project, the Klaytn format of signing hashes was applied.

The DAO in Wyvern v2.2 is based on a vote by an ERC20 share token. To make voting for small shareholders easier, the system allows the delegation of votes to a larger shareholder. Each proposal consists of the target address and contract calldata, along with the amount of ETH that will be sent alongside with it. The vote will pass if the minimum quorum is reached and more than half of the voters agree with it. It is also possible to change the voting rules via governance.

The DAO part of the contract was not used, and is out of the scope for audit.

Scope

```
├── Common
│   ├── ArrayUtils.sol
│   ├── ReentrancyGuarded.sol
│   ├── TokenLocker.sol
│   ├── TokenRecipient.sol
├── exchange
│   ├── Exchange.sol
│   ├── ExchangeCore.sol
│   ├── SaleKindInterface.sol
├── registry
    ├── proxy
    │   └── OwnedUpgradeabilityProxy.sol
    │   └── OwnedUpgradeabilityStorage.sol
    │   └── Proxy.sol
    ├── AuthenticatedProxy.sol
    ├── OwnableDelegateProxy.sol
    ├── ProxyRegistry.sol
    ├── TokenTransferProxy.sol
├── Migrations.sol
├── WyvernAtomicizer.sol
├── WyvernDAO.sol
├── WyvernDAOProxy.sol
├── WyvernExchange.sol
├── WyvernProxyRegistry.sol
├── WyvernTokenTransferProxy.sol
```

FINDINGS

1. Denial of Service can be occured since the contract “AuthenticatedProxy” is not initialized.

ID: MarbleX-01
Type: Denial of Service
File: contracts/WyvernRegistry.sol

Severity: Medium
Difficulty: Low

Issue

Any user can gain control and self-destruct a special *AuthenticatedProxy* contract. This may cause Denial of Service of trading at exchange services. This vulnerability was found and patched back in 2019, but was not patched in the project’s repository. See this link for details.

```
 constructor ()
    public
{
    delegateProxyImplementation = new AuthenticatedProxy();
}
```

[https://github.com/Bisonai-CIC/klaybay-wyvern-contracts/blob/develop/contracts/WyvernProxyRegistry.sol#L23]

Recommendation

Add initialization of *AuthenticatedProxy* will resolve this issue. Consult this commit.

2. The hashOrder related vulnerability is not patched, leads to potential theft of assets

ID: MarbleX-02	Severity: Critical
Type: Cryptography	Difficulty: Low
File: contracts/exchange/ExchangeCore.sol	

Issue

Following *contracts/exchange/Exchange.sol#L85*, the caller function of hashOrder function, gets *bytes calldata*, *bytes replacementPattern*, *bytes staticExtradata* as arguments, makes these as an Order object and pass to the hashOrder_ function. Since the bytes is a type of dynamic array, the hashOrder_ function must consider that the lengths of these inputs can be manipulated.

It was found that this vulnerability leads to asset theft. You can check more details at https://nft.mirror.xyz/VdF3BYwuzXgLrJglw5xF6CHcQfAVbqeJVtueCr4BUzs

```
function hashOrder_(
        address[7] addrs,
        uint256[9] uints,
        FeeMethod feeMethod,
        SaleKindInterface.Side side,
        SaleKindInterface.SaleKind saleKind,
        AuthenticatedProxy.HowToCall howToCall,
        bytes calldata,
        bytes replacementPattern,
        bytes staticExtradata
    ) public pure returns (bytes32) {
        return
            hashOrder(
                Order(
                    addrs[0],
                    addrs[1],
                    addrs[2],
                    uints[0],
                    uints[1],
                    uints[2],
                    uints[3],
                    addrs[3],
                    feeMethod,
                    side,
                    saleKind,
                    addrs[4],
                    howToCall,
                    calldata,
                    replacementPattern,
                    addrs[5],
                    staticExtradata,
                    ERC20(addrs[6]),
                    uints[4],
                    uints[5],
```

```
                    uints[6],
                    uints[7],
                    uints[8]
                )
            );
    }
```

```
function hashOrder(Order memory order)
        internal
        pure
        returns (bytes32 hash)
    {
        /* Unfortunately abi.encodePacked doesn't work here, stack size constraints. */
        uint size = sizeOf(order);
        bytes memory array = new bytes(size);
        uint index;
        assembly {
            index := add(array, 0x20)
        }
        index = ArrayUtils.unsafeWriteAddress(index, order.exchange);
        index = ArrayUtils.unsafeWriteAddress(index, order.maker);
        index = ArrayUtils.unsafeWriteAddress(index, order.taker);
        index = ArrayUtils.unsafeWriteUint(index, order.makerRelayerFee);
        index = ArrayUtils.unsafeWriteUint(index, order.takerRelayerFee);
        index = ArrayUtils.unsafeWriteUint(index, order.makerProtocolFee);
        index = ArrayUtils.unsafeWriteUint(index, order.takerProtocolFee);
        index = ArrayUtils.unsafeWriteAddress(index, order.feeRecipient);
        index = ArrayUtils.unsafeWriteUint8(index, uint8(order.feeMethod));
        index = ArrayUtils.unsafeWriteUint8(index, uint8(order.side));
        index = ArrayUtils.unsafeWriteUint8(index, uint8(order.saleKind));
        index = ArrayUtils.unsafeWriteAddress(index, order.target);
        index = ArrayUtils.unsafeWriteUint8(index, uint8(order.howToCall));
        index = ArrayUtils.unsafeWriteBytes(index, order.calldata);
        index = ArrayUtils.unsafeWriteBytes(index, order.replacementPattern);
        index = ArrayUtils.unsafeWriteAddress(index, order.staticTarget);
        index = ArrayUtils.unsafeWriteBytes(index, order.staticExtradata);
        index = ArrayUtils.unsafeWriteAddress(index, order.paymentToken);
        index = ArrayUtils.unsafeWriteUint(index, order.basePrice);
        index = ArrayUtils.unsafeWriteUint(index, order.extra);
        index = ArrayUtils.unsafeWriteUint(index, order.listingTime);
        index = ArrayUtils.unsafeWriteUint(index, order.expirationTime);
        index = ArrayUtils.unsafeWriteUint(index, order.salt);
        assembly {
            hash := keccak256(add(array, 0x20), size)
        }
        return hash;
    }
```

[https://github.com/Bisonai-CIC/klaybay-wyvern-contracts/blob/develop/contracts/exchange/ExchangeCore.sol#L250]

Recommendation

The best way to patch is to use EIP712. The bare minimum is to keccak256 hash the variable length values in the order before hashing and add chain ID to the hash as well.

3. Logical error in guardedArrayReplace function leads to memory overwriting

ID: MarbleX-03
Severity: High
Type: Low-Level (Memory Corruption)
Difficulty: High
File: contracts/WyvernRegistry.sol

Issue

guardedArrayReplace copies an extra word that is out of bounds of the bytes array when a length of bytes array is a multiple of 32. This leads to memory corruption.

```
function guardedArrayReplace(bytes memory array, bytes memory desired, bytes memory mask)
        internal
        pure
    {
        require(array.length == desired.length);
        require(array.length == mask.length);

        uint words = array.length / 0x20;
        uint index = words * 0x20;
        assert(index / 0x20 == words);
        uint i;

        for (i = 0; i < words; i++) {
            /* Conceptually: array[i] = (!mask[i] && array[i]) || (mask[i] && desired[i]), bitwise
in word chunks. */
            assembly {
                let commonIndex := mul(0x20, add(1, i))
                let maskValue := mload(add(mask, commonIndex))
                mstore(add(array, commonIndex), or(and(not(maskValue), mload(add(array,
commonIndex))), and(maskValue, mload(add(desired, commonIndex)))))
            }
        }

        /* Deal with the last section of the byte array. */
        if (words > 0) {
            /* This overlaps with bytes already set but is still more efficient than iterating
through each of the remaining bytes individually. */
            i = words;
            assembly {
                let commonIndex := mul(0x20, add(1, i))
                let maskValue := mload(add(mask, commonIndex))
                mstore(add(array, commonIndex), or(and(not(maskValue), mload(add(array,
commonIndex))), and(maskValue, mload(add(desired, commonIndex)))))
            }
        } else {
            /* If the byte array is shorter than a word, we must unfortunately do the whole thing
bytewise.
               (bounds checks could still probably be optimized away in assembly, but this is a rare
case) */
            for (i = index; i < array.length; i++) {
                array[i] = ((mask[i] ^ 0xff) & array[i]) | (mask[i] & desired[i]);
            }
        }
```

```
    }
```

[https://github.com/Bisonai-CIC/klaybay-wyvern-contracts/blob/develop/contracts/common/ArrayUtils.sol#L28]

Recommendation

The patch we recommend is to let the `commonIndex` value in the case where `words>0` to simply equal to the length of the array. In that case, the range of the bytes memory handled by the contract will be exactly the range of the bytes memory that is taken by the array variable.

Fix

Last Update: 2022.10.21

#ID	Title	Type	Severity	Difficulty	Status
1	Denial of Service can be occured since the contract “AuthenticatedProxy” is not initialized.	Denial of Service	Medium	Low	Fixed
2	The hashOrder related vulnerability is not patched, leads to potential theft of assets	Cryptography	Critical	Low	Fixed
3	Logical error in guardedArrayReplace function leads to memory overwriting	Low-Level	High	High	Fixed

Fix Comment

Issue 1 was patched in this pull request.

Issue 2 was patched in this pull request.

Issue 3 was patched in this pull request.

DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the main network. In order to write secure smart contracts, correction of discovered problems and sufficient testing thereof are required.

Appendix. A

Severity Level

CRITICAL	Must be addressed as a vulnerability that has the potential to seize or freeze substantial sums of money.
HIGH	Has to be fixed since it has the potential to deny users compensation or momentarily freeze assets.
MEDIUM	Vulnerabilities that could halt services, such as DoS and Out-of-Gas, need to be addressed.
LOW	Issues that do not comply with standards or return incorrect values
TIPS	Tips that makes the code more usable or efficient when modified

Difficulty Level

	Low	Medium	High
Privilege	anyone	Miner/Block Proposer	Admin/Owner
Capital needed	Small or none	Gas fee or volatile as price change	More than exploited amount
Probability	100%	Depend on environment	Hard as mining difficulty

Vulnerability Category

Arithmetic	• Integer under/overflow vulnerability • floating point and rounding accuracy
Access & Privilege Control	• Manager functions for emergency handle • Crucial function and data access • Count of calling important task, contract state change, intentional task delay
Denial of Service	• Unexpected revert handling • Gas limit excess due to unpredictable implementation
Miner Manipulation	• Dependency on the block number or timestamp. • Frontrunning
Reentrancy	• Proper use of Check-Effect-Interact pattern. • Prevention of state change after external call • Error handling and logging.
Low-level Call	• Code injection using delegatecall • Inappropriate use of assembly code
Off-standard	• Deviate from standards that can be an obstacle of interoperability.
Input Validation	• Lack of validation on inputs.
Logic Error/Bug	• Unintended execution leads to error.
Documentation	• Coherency between the documented spec and implementation
Visibility	• Variable and function visibility setting
Incorrect Interface	• Contract interface is properly implemented on code.

End of Document