

# HAECHI AUDIT

## BIFROST NFT Marketplace

Smart Contract Security Analysis

Published on : Jul 29, 2022

Version v2.0





# HAECHI AUDIT

Smart Contract Audit Certificate



## BIFROST NFT Marketplace

Security Report Published by HAECHI AUDIT  
v2.0 Jul 29, 2022

Auditor : Paul Kim, Allen Roh

### Executive Summary

Severity of Issues	Findings	Resolved	Unresolved	Acknowledged	Comment
Critical	1	1	-	-	-
Major	4	4	-	-	-
Minor	4	3	-	-	-
Tips	2	2	-	-	-

# TABLE OF CONTENTS

*9 Issues (1 Critical, 4 Major, 4 Minor) Found*

[TABLE OF CONTENTS](#)

[ABOUT US](#)

[INTRODUCTION](#)

[SUMMARY](#)

[OVERVIEW](#)

[FINDINGS](#)

[\[FrontEnd\] 프론트엔드가 필요하지 않은 서명을 요구하여, 공격자가 Replay 할 수 있습니다. \(Resolved - v.1.0\)](#)

[Issue](#)

[Recommendation](#)

[Update](#)

[\[Contract\] Wyvern v2.2의 Order Hashing 관련 취약점이 제대로 패치되지 않았습니다. \(Resolved - v.2.0\)](#)

[Issue](#)

[Recommendation](#)

[Update](#)

[Update 2](#)

[\[Contract\] FeeManagerLogic01 임의의 NFT의 정상적인 totalSupply\(\) 및 ownerOf\(\) 구현에 의존합니다 \(Resolved - v.2.0\)](#)

[Issue](#)

[Recommendation](#)

[Update](#)

[Update 2](#)

[\[Contract/Backend\] Referral Fee에 대한 로직이 취약하게 설계되어 Referral Fee를 탈취할 수 있는 취약점이 존재합니다. \(Resolved - v.2.0\)](#)

[Issue](#)

[Recommendation](#)

[Update](#)

[\[Contract\] Wyvern v2.2의 Denial of Service Vulnerability가 패치되지 않았습니다 \(Resolved - v.1.0\)](#)

[Issue](#)

[Update](#)

[\[Contract\] AuctionMatcher의 transferOwnership의 require 문이 잘못되었습니다. \(Resolved - v.2.0\)](#)

[Issue](#)

[Recommendation](#)

[Update](#)

[\[Contract\] ERC20 Transfer의 return 값을 확인하지 않고 있습니다. \(Found - v.2.0\)](#)

[Issue](#)

[Recommendation](#)

[Disclaimer](#)

[\[Out-of-Scope\] ImgProxy에서 임의의 서버를 대상으로 리퀘스트를 발생시킬 수 있습니다.](#)

[Issue](#)

[Recommendation](#)

[Update](#)

[\[Backend\] marketplace\\_apollo의 api에 외부 사이트에서 리퀘스트를 보낼 수 있습니다.](#)

[Issue](#)

[Recommendation](#)

[Update](#)

[\[Contract\] Wyvern v2.2는 이더리움 메인넷에서 USDT를 지원하지 않습니다. \(Resolved - v.1.0\)](#)

[Issue](#)

[Recommendation](#)

[Update](#)

[\[Contract\] Solidity Version을 고정하는 것을 추천합니다. \(Resolved - v.2.0\)](#)

[Issue](#)

[Recommendation](#)

[Update](#)

[DISCLAIMER](#)



# ABOUT US

---

HAECHI AUDIT은 디지털 자산이 가져올 금융 혁신을 믿습니다. 디지털 자산을 쉽고 안전하게 만들기 위해 HAECHI AUDIT은 '보안'과 '신뢰'라는 가치를 제공합니다. 그로써 모든 사람이 디지털 자산을 부담없이 활용할 수 있는 세상을 꿈꿉니다.

---

HAECHI AUDIT은 글로벌 블록체인 업계를 선도하는 HAECHI LABS의 대표 서비스 중 하나로, 스마트 컨트랙트 보안 감사 및 개발을 전문적으로 제공합니다.

다년간 블록체인 기술 연구 개발 경험을 보유하고 있는 전문가들로 구성되어 있으며, 그 전문성을 인정받아 블록체인 기술 기업으로는 유일하게 삼성전자 스타트업 육성 프로그램에 선정된 바 있습니다. 또한, 이더리움 재단과 이더리움 커뮤니티 펀드로부터 기술 장려금을 수여받기도 하였습니다.

대표적인 클라이언트 및 파트너사로는 카카오 자회사인 Ground X, LG, 한화, 신한은행 등이 있으며, Sushiswap, 1inch, Klaytn, Badger와 같은 글로벌 블록체인 프로젝트와도 협업한 바 있습니다. 지금까지 약 300여곳 이상의 클라이언트를 대상으로 가장 신뢰할 수 있는 스마트 컨트랙트 보안감사 및 개발 서비스를 제공하였습니다.

문의 : [audit@haechi.io](mailto:audit@haechi.io)

웹사이트 : [audit.haechi.io](https://audit.haechi.io)

# INTRODUCTION

---

본 보고서는 BIFROST 팀이 제작한 BMail NFT Marketplace 스마트 컨트랙트 및 웹 백엔드/프론트엔드의 보안을 감사하기 위해 작성되었습니다. HAECHI AUDIT 는 BIFROST 팀이 제작한 스마트 컨트랙트 및 웹 백엔드/프론트엔드의 구현 및 설계가 공개된 자료에 명시한 것처럼 잘 구현이 되어있고, 보안상 안전한지에 중점을 맞춰 감사를 진행했습니다.

---

## CRITICAL

Critical 이슈는 광범위한 사용자가 피해를 볼 수 있는 치명적인 보안 결점으로 반드시 해결해야 하는 사항입니다.

## MAJOR

Major 이슈는 보안상에 문제가 있거나 의도와 다른 구현으로 수정이 필요한 사항입니다.

## MINOR

Minor 이슈는 잠재적으로 문제를 발생시킬 수 있으므로 수정이 요구되는 사항입니다.

## TIPS

Tips 이슈는 수정했을 때 코드의 사용성이나 효율성이 더 좋아질 수 있는 사항입니다.












HAECHI AUDIT는 BIFROST 팀이 발견된 모든 이슈에 대하여 개선하는 것을 권장합니다.

이어지는 이슈 설명에서는 코드를 세부적으로 지칭하기 위해서 {파일 이름}#{줄 번호}, {컨트랙트 이름}#{함수/변수 이름} 포맷을 사용합니다. 예를 들면, *Sample.sol:20*은 Sample.sol 파일의 20번째 줄을 지칭하며, *Sample#fallback()* 는 Sample 컨트랙트의 fallback() 함수를 가리킵니다. 보고서 작성을 위해 진행된 모든 테스트 결과는 Appendix에서 확인 하실 수 있습니다.

# SUMMARY

Audit에 사용된 코드는 Github (Private Repository)에서 찾아볼 수 있습니다.  
(<https://github.com/HAECHI-LABS/BIFROST-NFT-Marketplace-Audit>)

**Issues** HAECHI AUDIT에서는 Critical 이슈 1개, Major 이슈 4개, Minor 이슈 4개를 발견하였으며 수정했을 때 코드의 사용성이나 효율성이 더 좋아질 수 있는 사항들을 2개의 Tips 카테고리로 나누어 서술하였습니다.

Severity	Issue	Status
 <b>CRITICAL</b>	프런트엔드가 필요하지 않은 서명을 요구하여, 공격자가 Replay 할 수 있습니다.	(Resolved - v1.0)
 <b>MAJOR</b>	Wyvern v2.2의 Order Hashing 관련 취약점이 제대로 패치되지 않았습니다.	(Resolved - v2.0)
 <b>MAJOR</b>	FeeManagerLogic이 임의의 NFT의 정상적인 totalSupply() 및 ownerOf() 구현에 의존합니다	(Resolved - v2.0)
 <b>MAJOR</b>	Referral Fee에 대한 로직이 전반적으로 취약해 Referral Fee를 탈취할 수 있습니다.	(Resolved - v2.0)
 <b>MAJOR</b>	Wyvern v2.2의 Denial of Service Vulnerability가 패치되지 않았습니다	(Resolved - v1.0)
 <b>MINOR</b>	AuctionMatcher의 transferOwnership의 require 문이 잘못되었습니다.	(Resolved - v2.0)
 <b>MINOR</b>	ERC20 Transfer의 return 값을 확인하지 않고 있습니다.	(Found - v2.0)
 <b>MINOR</b>	ImgProxy에서 임의의 서버를 대상으로 리퀘스트를 발생시킬 수 있습니다.	(Resolved - v2.0)
 <b>MINOR</b>	marketplace_apollo의 api에 외부 사이트에서 리퀘스트를 보낼 수 있습니다.	(Resolved - v2.0)
 <b>TIPS</b>	Wyvern v2.2는 이더리움 메인넷에서 USDT를 지원하지 않습니다.	(Resolved - v1.0)
 <b>TIPS</b>	Solidity Version을 고정하는 것을 추천합니다.	(Resolved - v2.0)



# OVERVIEW

## Web subject to audit

- ❖ Backend/marketplace-apollo
- ❖ Backend/marketplace-eclipse
- ❖ Backend/marketplace-luna
- ❖ Frontend

## Contracts subject to audit

- ❖ AuctionMatcher
- ❖ BatchLogic
- ❖ BatchProxy
- ❖ CallProxy
- ❖ ArrayUtils
- ❖ ReentrancyGuarded
- ❖ TokenLocker
- ❖ TokenRecipient
- ❖ Exchange
- ❖ ExchangeCore
- ❖ SaleKindInterface
- ❖ FeeManagerLogic
- ❖ FeeManagerProxy
- ❖ IERC20
- ❖ IERC721
- ❖ IFeeManager
- ❖ IFeeManagerProxy
- ❖ AuthenticatedProxy
- ❖ OwnableDelegateProxy
- ❖ ProxyRegistry
- ❖ TokenTransferProxy
- ❖ OwnedUpgradeabilityProxy
- ❖ OwnedUpgradeabilityStorage
- ❖ Proxy
- ❖ Bmath
- ❖ Owner
- ❖ WyvernExchange
- ❖ WyvernProxyRegistry
- ❖ WyvernTokenTransferProxy

BIFROST NFT Marketplace Smart contract에는 다음과 같은 권한이 있습니다.

- ❖ Owner
- ❖ Matcher
- ❖ ProxyOwner


각 권한의 제어에 대한 명세는 다음과 같습니다.

Owner는 FeeManager에서 NFT의 whitelist부터 fee 관련 각종 값들까지 모두 변경할 수 있는 강력한 권한을 가지고 있습니다. 하지만, Owner의 강력한 권한은 대부분 유저의 자산 자체를 제어하는 것이 아닌 Fee 로직에 대한 제어 권한입니다. 중앙화 이슈가 다소 존재하여 Owner의 private key 가 잘못 관리되어 탈취되거나 Owner가 잘못된 트랜잭션을 발생시키는 행동 등을 하게 된다면 유저들이 받아야 하는 수수료를 받지 못하는 상황이 생길 수 있으며 이런 상황은 이 보고서에서 가정하지 않는 상황입니다. 하지만 유저의 NFT, 결제대금에 대해 Owner가 제어하지는 않습니다.

Role	Functions
Owner	<ul style="list-style-type: none"><li>❖ <i>AuctionMatcher#setMatcher()</i></li><li>❖ <i>AuctionMatcher#transferOwnership()</i></li><li>❖ <i>BatchTransfer#tokenBatchs()</i></li><li>❖ <i>BatchTransfer#nftBatchs()</i></li><li>❖ <i>BatchProxy#setImplementation()</i></li><li>❖ <i>CallProxy#setFeeManager()</i></li><li>❖ <i>CallProxy#setOwner()</i></li><li>❖ <i>TokenLocker#transfer()</i></li><li>❖ <i>ExchangeCore#changeMinimumMakerProtocolFee()</i></li><li>❖ <i>ExchangeCore#changeMinimumTakerProtocolFee()</i></li><li>❖ <i>ExchangeCore#changeProtocolFeeRecipient()</i></li><li>❖ <i>FeeManagerProxy#setImplementation()</i></li><li>❖ <i>FeeManagerLogic#communityFeeUpdate()</i></li><li>❖ <i>FeeManagerLogic#adminWithdraw()</i></li><li>❖ <i>FeeManagerLogic#setOwner()</i></li><li>❖ <i>FeeManagerLogic#setWyvernProtocolAddr()</i></li><li>❖ <i>FeeManagerLogic#setNFTTotalSupply()</i></li><li>❖ <i>FeeManagerLogic#pause()</i></li><li>❖ <i>FeeManagerLogic#unpause()</i></li><li>❖ <i>FeeManagerLogic#setWhiteList()</i></li><li>❖ <i>FeeManagerLogic#setClaimedCommunityFeeInCollection()</i></li><li>❖ <i>FeeManagerLogic#setClaimedCommunityFee()</i></li><li>❖ <i>FeeManagerLogic#setAccumulatedCommunityFee()</i></li><li>❖ <i>ProxyRegistry#startGrantAuthentication()</i></li><li>❖ <i>ProxyRegistry#endGrantAuthentication()</i></li><li>❖ <i>ProxyRegistry#revokeAuthentication()</i></li></ul>

Matcher	❖ <i>AuctionMatcher#isMatcher()</i>
ProxyOwner	❖ <i>OwnedUpgradeabilityProxy#transferProxyOwnership()</i> ❖ <i>OwnedUpgradeabilityProxy#upgradeTo()</i> ❖ <i>OwnedUpgradeabilityProxy#upgradeToAndCall()</i>

# FINDINGS

 **CRITICAL** [FrontEnd] 프론트엔드가 필요하지 않은 서명을 요구하여, 공격자가 Replay 할 수 있습니다. (Resolved - v.1.1.0)

```
{
  title: t("step-checkout"),
  description: t("step-checkout-desc"),
  call: async () => {
    let salePayment = getBMallTokenByContractAddress(
      saleOrder.chainID,
      saleOrder.paymentToken
    );
    let newOrder = await getAndSignOrder(
      nft.chainID,
      nft.contractAddress,
      nft.tokenID,
      saleOrder.basePrice.toLocaleString("fullwide", {
        useGrouping: false,
      }),
      1 - saleOrder.side,
      walletAddress,
      0,
      salePayment?.paymentType,
      saleOrder.takerRelayerFee != 0,
      saleOrder.takerRelayerFee != 0 && saleOrder.side === OrderSide.Buy
    );

    let tx = await atomicMatch(
      contractInfo.Exchange,
      saleOrder.side === OrderSide.Sell ? saleOrder : newOrder,
      saleOrder.side === OrderSide.Sell ? newOrder : saleOrder,
      walletAddress
    );
    if (tx) {
      await tx.wait();
    }
    await sleep(3000);
    await mutate(
      `/order?contractID=${nft.contractID}&tokenID=${nft.tokenID}`
    );
    await mutate(
      `/nft/history?contractID=${nft.contractID}&tokenID=${nft.tokenID}`
    );
    await mutate(
      `/nft/sale?contractID=${nft.contractID}&tokenID=${nft.tokenID}`
    );
  },
},
```

## Issue

Wyvern 2.2의 `atomicMatch`는 기본적으로 구매자의 Order와 그 서명, 판매자의 Order와 그 서명을 입력으로 받아서 작동합니다. 서명은 기본적으로 Order를 실제로 구매자/판매자가 승인을

했는지 판단하는 용도로 사용됩니다. 그런데 구매자가 트랜잭션을 보낸 사람이라면, 애초에 구매자가 트랜잭션을 서명했음에니 구매자의 Order를 따로 서명할 이유가 없습니다. 그래서 이 경우에 Wyvern 2.2 컨트랙트는 구매자의 서명을 따로 검사하지 않으며, Signature Replay를 막기 위해서 구매자의 Order 해시가 사용되었다는 체크도 따로 진행하지 않습니다.

```
bytes32 buyHash;
if (buy.maker == msg.sender) {
    require(validateOrderParameters(buy));
} else {
    buyHash = requireValidOrder(buy, buySig);
}

/* Ensure sell order validity and calculate hash if necessary. */
bytes32 sellHash;
if (sell.maker == msg.sender) {
    require(validateOrderParameters(sell));
} else {
    sellHash = requireValidOrder(sell, sellSig);
}
```

```
/* Mark previously signed or approved orders as finalized. */
if (msg.sender != buy.maker) {
    cancelledOrFinalized[buyHash] = true;
}
if (msg.sender != sell.maker) {
    cancelledOrFinalized[sellHash] = true;
}
```

그래서 OpenSea의 경우, 구매자가 트랜잭션을 보낸 사람인 경우 구매자의 서명을 요구하지 않으며, 트랜잭션을 보낼 때 판매자의 서명을 그대로 구매자의 서명으로 사용합니다.

아래 사진은 OpenSea Wyvern Exchange에서 일어난 한 거래의 **rssMetadata** 값입니다. **(r, s)**의 쌍이 구매자의 서명과 판매자의 서명에서 일치함을 확인할 수 있습니다.

```
bytes32[5] 0x3e290e7de4d1fdbd86a2b4aa4bf395379a54e47065f07801bc4be459af1bf614
           0x700656772c03d5a6f92419f2a76c6478b9d40d2eec41d84e47e97ee761c445ea
           0x3e290e7de4d1fdbd86a2b4aa4bf395379a54e47065f07801bc4be459af1bf614
           0x700656772c03d5a6f92419f2a76c6478b9d40d2eec41d84e47e97ee761c445ea
           0x0000000000000000000000000000000000000000000000000000000000000000
```

<https://etherscan.io/tx/0x0b034cfa594df00bed870792fa7c55d7363b0d287ac42ca04c12231ee5f72304>

현재 프론트엔드는 이와 무관하게 서명을 항상 요구하고 있습니다. 이 서명은 컨트랙트에서 사용되지 않음에도 불구하고 트랜잭션 데이터에 포함되어 있기 때문에 모두에게 공개되어 있으며, 아무나 이 서명을 사용하여 사용자가 원하지 않는 거래를 강제로 체결할 수 있습니다.

### Recommendation

*atomicMatch*를 호출하는 주소에게 하는 Signature 생성 요청을 제거해주세요. 추가적으로, 주문을 취소하는 경우 항상 `cancelOrder`를 스마트 컨트랙트에 제대로 호출하고, 호출이 실패했을 경우에는 취소처리가 제대로 실패하는지 다시 한 번 확인하는 것을 추천합니다.

### Update

BIFROST 측에서 필요하지 않은 Signature 생성 요청을 제거하였습니다.

**MAJOR** [Contract] Wyvern v2.2의 Order Hashing 관련 취약점이 제대로 패치되지 않았습니다. (Resolved - v.2.0)

```
/**
 * Calculate size of an order struct when tightly packed
 *
 * @param order Order to calculate size of
 * @return Size in bytes
 */
function sizeOf(Order memory order)
    internal
    pure
    returns (uint)
{
    return ((0x14 * 7) + (0x20 * 9) + 4 + order.calldata.length +
order.replacementPattern.length + order.staticExtradata.length);
}

/**
 * @dev Hash an order, returning the canonical order hash, without the message prefix
 * @param order Order to hash
 * @return Hash of order
 */
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.unsafeWriteAddressWord(index, order.exchange);
    index = ArrayUtils.unsafeWriteAddressWord(index, order.maker);
    index = ArrayUtils.unsafeWriteAddressWord(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.unsafeWriteAddressWord(index, order.feeRecipient);
    index = ArrayUtils.unsafeWriteUint8Word(index, uint8(order.feeMethod));
    index = ArrayUtils.unsafeWriteUint8Word(index, uint8(order.side));
    index = ArrayUtils.unsafeWriteUint8Word(index, uint8(order.saleKind));
    index = ArrayUtils.unsafeWriteAddressWord(index, order.target);
    index = ArrayUtils.unsafeWriteUint8Word(index, uint8(order.howToCall));
    index = ArrayUtils.unsafeWriteBytes32(index, keccak256(order.calldata));
    index = ArrayUtils.unsafeWriteBytes32(index, keccak256(order.replacementPattern));
    index = ArrayUtils.unsafeWriteAddressWord(index, order.staticTarget);
    index = ArrayUtils.unsafeWriteBytes32(index, keccak256(order.staticExtradata));
    index = ArrayUtils.unsafeWriteAddressWord(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;
}
```

## Issue

최근 Wyvern 2.2에서 Order Signing 과정에서 발견된 Critical 한 취약점이 있었습니다.

<https://nft.mirror.xyz/VdF3BYwuzXgLrJglw5xF6CHcOfAVbqeJVtueCr4BUzs>

BIFROST에서 이를 패치하였는데, 여전히 문제가 있습니다. 우선 *calldata, replacementPattern, staticExtradata*가 keccak256 해시 처리가 되었음에도 불구하고, *sizeOf*는 여전히 이 bytes 들의 길이를 사용하고 있습니다. 이에 따라서 bytes의 사이즈가 충분히 길지 않은 경우가 발생할 수 있고, 이에 따라서 다른 두 Order가 같은 값으로 해시되는 경우가 발생할 수 있습니다.

또한, Order를 서명할 때 들어가는 정보에서 chainID가 빠져있어, 서로 다른 두 블록체인에서 같은 Order 및 그 서명이 유효할 수 있습니다. 실제로 이 시나리오가 발생하려면 exchange address와 NFT address가 두 블록체인에서 겹치면 되는데, 충분히 가능한 상황으로 보입니다.

## Recommendation

*sizeOf* 함수를 수정하고, Order를 서명하는 과정에서 chainID를 추가하는 것을 제안합니다. 나아가서, 가능하면 EIP712 스펙에 맞게 Order 해싱을 구현하는 것을 제안합니다.

## Update

BIFROST 측에서는 사이즈를 다시 계산하고, chainID를 uint8로 추가했습니다.

보안적인 이슈는 resolve가 되었으나, EIP712 스펙에 맞게 chainID를 uint8이 아닌 uint256으로 추가하고, 이 값을 실제 블록체인의 ID와 일치시키는 것을 추천합니다. uint8 혼란을 막기 위해서, 더불어 ApolloQL에 있는 chainID도 실제 블록체인의 ID와 일치시키는 것을 제안합니다. 또한, 가능하면 EIP712 스펙을 전부 따라가는 방식으로 (*domainSeparator* 등 추가) 주문을 해싱하는 로직을 구현하는 것을 추천합니다.

## Update 2

BIFROST 측에서는 chainID를 uint256으로 변경했습니다.



## ⚠ MAJOR

[Contract] FeeManagerLogic이 임의의 NFT의 정상적인 totalSupply() 및 ownerOf() 구현에 의존합니다 (Resolved - v.2.0)

```
function _feeClaim(address nftAddr, uint256[] memory tokenID, address[] memory tokenAddrs) internal
{
    IERC721 nft = IERC721(nftAddr);

    CommunityFeeParams memory communityFeeParams;
    communityFeeParams.totalSupply = nft.totalSupply() * UNIFIEDPOINT;
    communityFeeParams.tokenLength = tokenID.length * UNIFIEDPOINT;

    // This code for blocking nft's minting. if specific nft is minted in Bmall, maybe this nft is
    blocked.
    if(nftTotalSupply[nftAddr] == 0){
        nftTotalSupply[nftAddr] = communityFeeParams.totalSupply;
    }
    require( nftTotalSupply[nftAddr] == communityFeeParams.totalSupply, "NFT totalSupply is
    changed");
    //

    uint256[] memory rewardAmount = new uint256[](tokenAddrs.length);

    for(uint256 tokenAddrIndex = 0; tokenAddrIndex < tokenAddrs.length; tokenAddrIndex++){
        address _tokenAddr = tokenAddrs[tokenAddrIndex];

        for(uint256 tokenIDIndex = 0; tokenIDIndex < tokenID.length; tokenIDIndex++) {
            uint256 _tokenID = tokenID[tokenIDIndex];
            address nftOwner = nft.ownerOf(_tokenID);
            require(nftOwner == msg.sender, "Do not match nft owners");

            communityFeeParams.rewardPerNFT =
            accumulatedCommunityFee[nftAddr][_tokenAddr].unifiedDiv(communityFeeParams.totalSupply);

            if(communityFeeParams.rewardPerNFT >
            claimedCommunityFee[nftAddr][_tokenAddr][_tokenID]){
                communityFeeParams.rewardPerNFT -=
            claimedCommunityFee[nftAddr][_tokenAddr][_tokenID];
            }else{
                continue;
            }

            claimedCommunityFee[nftAddr][_tokenAddr][_tokenID] += communityFeeParams.rewardPerNFT;
            rewardAmount[tokenAddrIndex] += communityFeeParams.rewardPerNFT;
        }
    }

    for(uint256 tokenIndex=0; tokenIndex < tokenAddrs.length; tokenIndex++){
        if(tokenAddrs[tokenIndex] == NATIVECOINADDR){
            payable(msg.sender).transfer(rewardAmount[tokenIndex]);
        }else{
            IERC20 token = IERC20(tokenAddrs[tokenIndex]);
            uint256 underlyingDecimal = uint256(10 ** token.decimals());
            uint256 underlyingAmount =
            rewardAmount[tokenIndex].unifiedToUnderlyingAmount(underlyingDecimal);
            token.transfer(msg.sender, underlyingAmount);
        }
    }
}
```

```
emit FeeClaim(nftAddr, tokenID, tokenAddrs, rewardAmount);
}
```

## Issue

FeeManagerLogic Contract에서는 Community Fee를 관리하고 이를 NFT Holder들에게 분배하는 로직을 작성하고 있습니다. 이 과정에서 NFT의 `totalSupply()` 함수와 `ownerOf()` 함수를 사용하고 있는데, ExchangeCore Contract에서는 임의의 NFT를 거래할 수 있기 때문에 두 함수가 정상적으로 작동하지 않는 NFT도 거래할 수 있습니다. 이 점을 이용하여, 다음과 같이 FeeManagerProxy Contract에 있는 모든 Community Fee를 탈취할 수 있습니다.

FeeManagerProxy에 10000 ETH가 있다고 가정합니다. 공격자의 주소를 A라고 합시다.

1. `totalSupply()`가 1을 return 하고, `ownerOf()`가 A를 return 하는 malicious NFT contract를 공격자가 deploy 합니다.
2. 공격자가 100 ETH를 보유하고 있다고 합시다.
3. malicious NFT 하나를 Protocol Fee 100%로 100 ETH로 자가거래합니다. 이러면 해당 malicious NFT에 대응되는 `accumulatedCommunityFee`가 100 ETH가 됩니다.
4. 이제 `nftAddr`를 malicious NFT의 주소로, `tokenID`를 1부터 100까지의 자연수를 나열한 array로, `tokenAddrs`를 `[NATIVECOINADDR]`로 두고 `feeClaim()`를 호출합니다.
5. `nftTotalSupply[nftAddr]`가 0이기 때문에 total supply 관련 require 문이 통과하며, `ownerOf()`가 항상 A를 return 하기 때문에 owner 관련 require 문도 통과합니다. `totalSupply()`가 1을 return 하기 때문에 `rewardPerNFT`가 100 ETH가 되지만, 실제로는 100개의 NFT를 보유하고 있기 때문에 공격자는 10000 ETH를 전부 탈취하게 됩니다.
6. 2-5까지의 과정을 여러 토큰에 대해서 반복하면 모든 자산을 탈취할 수 있습니다. 이 과정은 반복 가능하며, 공격자의 자산이 부족한 경우 Flash Loan을 이용할 수 있습니다.

별개로 `totalSupply()`는 EIP721 상 Optional 이므로, NFT가 이를 구현하지 않을 수도 있습니다.

## Recommendation

`FeeManagerLogic` 을 적절히 수정하시기 바랍니다.

NFT에 대한 whitelist를 owner가 관리할 수 있도록 하는 것을 추천합니다. 이렇게 했을 때, NFT를 whitelist에 추가할 때 확인해야 하는 점을 대표적으로 몇 가지 나열하자면

1. Proxy Pattern을 사용하여 Upgrade가 가능하면 안됩니다.
2. `totalSupply()`와 `ownerOf()` 이 정상적으로 작동해야 합니다.
3. `totalSupply()`가 더 증가하지 않는 것이 (즉, 추가 mint가 불가능) 확인되어야 합니다.
4. `totalSupply()`가 더 감소하지 않는 것이 (즉, burn이 불가능) 확인되어야 합니다.

확인해야 하는 점은 물론 이에 국한되지는 않으며, 해당 NFT에 대한 보안적 측면을 전반적으로 고려해야 합니다. 그 후 NFT가 whitelist에 속하는지를 `feeClaim()`에서 확인하면 됩니다.

또 다른 방법은 오프체인에서 직접 각 사용자가 받아야 하는 Protocol Fee의 양을 계산한 후, 이를 MerkleDistributor 등을 사용하여 중앙화된 방식으로 분배하는 것입니다.

## Update

이 문제를 제보한 뒤, BIFROST 측은 다음과 같이 컨트랙트를 수정했습니다.

```
function _feeClaim(address nftAddr, uint256[] memory tokenID, address[] memory tokenAddrs) internal
{
    IERC721 nft = IERC721(nftAddr);

    CommunityFeeParams memory communityFeeParams;
    communityFeeParams.totalSupply = nft.totalSupply() * UNIFIEDPOINT;
    communityFeeParams.tokenLength = tokenID.length * UNIFIEDPOINT;

    // This code for blocking nft's minting. if specific nft is minted in Bmall, maybe this nft is
    blocked.
    if(nftTotalSupply[nftAddr] == 0){
        nftTotalSupply[nftAddr] = communityFeeParams.totalSupply;
    }
    require( nftTotalSupply[nftAddr] == communityFeeParams.totalSupply, "NFT totalSupply is
    changed"); // owner has to call set total supply every time there's a mint? -> this is intended.

    uint256[] memory rewardAmount = new uint256[](tokenAddrs.length);

    for(uint256 tokenAddrIndex = 0; tokenAddrIndex < tokenAddrs.length; tokenAddrIndex++){
        address _tokenAddr = tokenAddrs[tokenAddrIndex];

        for(uint256 tokenIDIndex = 0; tokenIDIndex < tokenID.length; tokenIDIndex++) {
            uint256 _tokenID = tokenID[tokenIDIndex];
            address nftOwner = nft.ownerOf(_tokenID);
            require(nftOwner == msg.sender, "Do not match nft owners");

            communityFeeParams.rewardPerNFT =
            accumulatedCommunityFee[nftAddr][_tokenAddr].unifiedDiv(communityFeeParams.totalSupply);
```

```

        if (communityFeeParams.rewardPerNFT >
claimedCommunityFee[nftAddr][_tokenAddr][_tokenId]){
            communityFeeParams.rewardPerNFT -=
claimedCommunityFee[nftAddr][_tokenAddr][_tokenId];
        } else {
            continue;
        }

        claimedCommunityFee[nftAddr][_tokenAddr][_tokenId] += communityFeeParams.rewardPerNFT;
        claimedCommunityFeeInCollection[nftAddr][_tokenAddr] += communityFeeParams.rewardPerNFT;
        rewardAmount[tokenAddrIndex] += communityFeeParams.rewardPerNFT;
    }
    // change here
    require(claimedCommunityFeeInCollection[nftAddr][_tokenAddr] <=
accumulatedCommunityFee[nftAddr][_tokenAddr], "Over claimed fees");

    if (tokenAddrs[tokenAddrIndex] == NATIVECOINADDR){
        payable(msg.sender).transfer(rewardAmount[tokenAddrIndex]);
    } else {
        IERC20 token = IERC20(tokenAddrs[tokenAddrIndex]);
        uint256 underlyingDecimal = uint256(10 ** token.decimals());
        uint256 underlyingAmount =
rewardAmount[tokenAddrIndex].unifiedToUnderlyingAmount(underlyingDecimal);
        token.transfer(msg.sender, underlyingAmount);
    }
}
emit FeeClaim(nftAddr, tokenId, tokenAddrs, rewardAmount);
}

```

즉, 실제로 claim 되는 Community Fee가 accumulate 되었던 Community Fee 이하라는 require 문을 추가하는 방식으로 컨트랙트를 수정했습니다.

이 경우, 특정 NFT에 대해서 Community Fee를 분배받지 못하도록 하는 Denial of Service 공격이 가능할 수 있습니다. 다음과 같은 사례를 생각해봅시다.

정상적인 NFT가 있고, 총 100개의 NFT가 민팅될 예정이라고 합시다.

1. 정상적인 NFT의 mint가 열리자마자, 공격자 A는 다음 과정을 거칩니다.
2. NFT를 하나 mint 하고, Protocol Fee 100%로 NFT를 100 ETH에 자가거래 합니다.
3. 바로 `feeClaim()`을 호출하여 100 ETH를 돌려받습니다.

이제 `accumulatedCommunityFee`는 100 ETH며, `claimedCommunityFee` 역시 100 ETH입니다. NFT 100개가 모두 민팅되고 `nftTotalSupply`가 100에 해당하는 값으로 변경되었다고 합시다. 다른 정상적인 사용자가 `feeClaim()`을 하려고 하면, `rewardPerNFT`의 값이 최소  $100 \text{ ETH} / 100 = 1 \text{ ETH}$ 가 됩니다. 새로 추가된 require 문을 통과하려면 최소한 새로

accumulate 된 Community Fee가 1 ETH는 되어야 한다는 의미이며, 이때까지 각 사용자들은 자신의 정당한 Community Fee를 Claim 할 수 없습니다. 이 문제는 Total Supply가 변화하는 상황에서 Community Fee를 각 NFT가 얼마나 분배받아야 하는지에 대한 계산이 제대로 되고 있지 않아서 발생합니다. 우선 아래 부분을 삭제하는 것을 추천합니다.

```
// This code for blocking nft's minting. if specific nft is minted in Bmall, maybe this nft is blocked.  
if(nftTotalSupply[nftAddr] == 0){  
    nftTotalSupply[nftAddr] = communityFeeParams.totalSupply;  
}
```

또한, 온체인에서 Community Fee의 분배를 하는 것이 목적이라면 Total Supply가 변화하는 경우에도 제대로 계산을 할 수 있는 로직을 고안하여 컨트랙트를 새로 작성하는 것을 추천합니다.

예를 들어, Community Fee를 분배받고 싶은 사람은 해당 NFT 및 tokenID를 등록하게 하고, Staking Reward를 분배하는 로직과 비슷한 방식으로 각 등록된 NFT에 대해 분배할 Community Fee의 양을 계산하는 것도 온체인에서 Community Fee를 분배하는 방법입니다.

이와 별개로, 온체인에서 Community Fee를 현재 방식으로 분배받는 경우 NFT Flash Loan을 통해서 Community Fee를 대신 받는 등의 다양한 가능성도 있음을 고려해야 합니다.

## Update 2

BIFROST는 다음과 같이 컨트랙트를 수정했습니다.

```
function _feeClaim(address nftAddr, uint256[] memory tokenID, address[] memory tokenAddrs) internal  
{  
    // mitigation for flashloan attack based on NFT  
    require(msg.sender == tx.origin);  
  
    // mitigation for malicious nft contract, add whitelist require statement  
    require(whitelist[nftAddr] == true, "only whitelist");  
  
    IERC721 nft = IERC721(nftAddr);  
  
    CommunityFeeParams memory communityFeeParams;  
    communityFeeParams.totalSupply = _getTotalSupply(nftAddr);  
    communityFeeParams.tokenLength = tokenID.length * UNIFIEDPOINT;  
  
    // This code for blocking nft's minting. if specific nft is minted in Bmall, maybe this nft is blocked.
```

```

    if(nftTotalSupply[nftAddr] == 0){
        nftTotalSupply[nftAddr] = communityFeeParams.totalSupply;
    }
    require( nftTotalSupply[nftAddr] == communityFeeParams.totalSupply, "NFT totalSupply is
changed");
    //

    uint256[] memory rewardAmount = new uint256[](tokenAddrs.length);

    for(uint256 tokenAddrIndex = 0; tokenAddrIndex < tokenAddrs.length; tokenAddrIndex++){
        address _tokenAddr = tokenAddrs[tokenAddrIndex];

        for(uint256 tokenIDIndex = 0; tokenIDIndex < tokenID.length; tokenIDIndex++) {
            uint256 _tokenID = tokenID[tokenIDIndex];
            address nftOwner = nft.ownerOf(_tokenID);
            require(nftOwner == msg.sender, "Do not match nft owners");

            communityFeeParams.rewardPerNFT =
            accumulatedCommunityFee[nftAddr][_tokenAddr].unifiedDiv(communityFeeParams.totalSupply);

            if(communityFeeParams.rewardPerNFT >
            claimedCommunityFee[nftAddr][_tokenAddr][_tokenID]){
                communityFeeParams.rewardPerNFT -=
            claimedCommunityFee[nftAddr][_tokenAddr][_tokenID];
            }else{
                continue;
            }

            claimedCommunityFee[nftAddr][_tokenAddr][_tokenID] += communityFeeParams.rewardPerNFT;
            claimedCommunityFeeInCollection[nftAddr][_tokenAddr] += communityFeeParams.rewardPerNFT;
            rewardAmount[tokenAddrIndex] += communityFeeParams.rewardPerNFT;
        }

        require(claimedCommunityFeeInCollection[nftAddr][_tokenAddr] <=
            accumulatedCommunityFee[nftAddr][_tokenAddr], "Over claimed fees");

        if(rewardAmount[tokenAddrIndex] > 0){
            if(tokenAddrs[tokenAddrIndex] == NATIVECOINADDR){
                payable(msg.sender).transfer(rewardAmount[tokenAddrIndex]);
            }else{
                IERC20 token = IERC20(tokenAddrs[tokenAddrIndex]);
                uint256 underlyingDecimal = uint256(10 ** token.decimals());
                uint256 underlyingAmount =
            rewardAmount[tokenAddrIndex].unifiedToUnderlyingAmount(underlyingDecimal);
                token.transfer(msg.sender, underlyingAmount);
            }
        }
    }

    emit FeeClaim(nftAddr, tokenID, tokenAddrs, rewardAmount);
}

// mitigation of totalSupply function not existed in erc721
function _getTotalSupply(address nftAddr) internal view returns (uint256) {
    IERC721 erc721 = IERC721(nftAddr);

    try erc721.totalSupply() returns (uint256 _value) {
        return (_value * UNIFIEDPOINT);
    }
    catch {

```

```

        require(nftTotalSupply[nftAddr] != 0, "Err: nft totalSupply is 0");
        return nftTotalSupply[nftAddr];
    }
}

function setWhiteList(address _nftAddr, bool _state) external onlyOwner {
    whitelist[_nftAddr] = _state;
    emit SetWhiteList(_nftAddr, _state);
}

function setClaimedCommunityFeeInCollection(address _nftAddr, address _tokenAddr, uint256
_claimedAmount) external onlyOwner {
    claimedCommunityFeeInCollection[_nftAddr][_tokenAddr] = _claimedAmount;
}

function setClaimedCommunityFee(address _nftAddr, address _tokenAddr, uint256 _nftID, uint256
_claimedAmount) external onlyOwner {
    claimedCommunityFee[_nftAddr][_tokenAddr][_nftID] = _claimedAmount;
}

function setAccumulatedCommunityFee(address _nftAddr, address _tokenAddr, uint256
_accumulatedAmount) external onlyOwner {
    accumulatedCommunityFee[_nftAddr][_tokenAddr] = _accumulatedAmount;
}

function setOwner(address _owner) external onlyOwner {
    owner = _owner;
}

function setWyvernProtocolAddr(address _wyvernProtocolAddr) external onlyOwner {
    wyvernProtocolAddr = _wyvernProtocolAddr;
}

function getWhiteList(address _nftAddr) external view returns (bool) {
    return whitelist[_nftAddr];
}

```

BIFROST 측은 whitelist를 추가하는 방식으로 문제를 해결하기로 결정했습니다. whitelist 변수들 및 communityFee를 관리하는 변수들까지 전부 관리자 권한으로 바꿀 수 있기 때문에, owner 권한을 갖는 private key의 관리에 각별한 주의가 필요합니다. 또한, whitelist를 관리하는 과정에서 각 whitelist 된 NFT가 보안적인 문제를 발생시키지 않는지도 검토해야 합니다.

**MAJOR** [Contract/Backend] Referral Fee에 대한 로직이 취약하게 설계되어 Referral Fee를 탈취할 수 있는 취약점이 존재합니다. (Resolved - v.2.0)

```
private async processWyvernExchangeOrdersMatched(logs: IDecodedParameterLogs[],
transferLogs: IDecodedParameterLogs[], nftContract: INFTContract, commonData:
IApplicationEventInfoData): Promise<void> {
  const abstractedTransferEventParameters =
this.abstractTransferEventParameters(transferLogs);
  const sender: string = abstractedTransferEventParameters.sender;
  const receiver: string = abstractedTransferEventParameters.receiver;
  const contractID: string = nftContract._id.toString();
  const tokenID: number = abstractedTransferEventParameters.tokenID;

  ...

  const signedOrders = await this.apolloQLHandler.findSignedOrders(
    nft._id.toString(),
    abstractedOrdersMatchedEventParameters.buyHash,
    abstractedOrdersMatchedEventParameters.sellHash,
  );

  if (signedOrders.length === 0) {
    logger.error(`SignedOrders cannot find ${nftContract._id.toString()} tokenID:
${tokenID}, sender: ${sender}`);
    return;
  }

  ...

  // creatorFee, marketFee has decimal
  const PERCENT_NUM = 10000;
  const creatorFee = Math.floor(tradePrice * collection.creatorFee / PERCENT_NUM);
  const marketFee = Math.floor(tradePrice * collection.serviceFee / PERCENT_NUM);

  const communityFee = Math.floor(tradePrice * collection.communityFee / nftCount);

  await this.apolloQLHandler.increaseCreatorFee(
    ...
  );

  ...

  // inc sender recommender referral fee
  const senderReferral = await this.apolloQLHandler.findOneReferral(sender);
  if (senderReferral !== null) {
    await this.apolloQLHandler.increaseReferralFee(
      ...
    );
  }

  // inc buyer recommender referral fee
  const buyerReferral = await this.apolloQLHandler.findOneReferral(receiver);
  if (buyerReferral !== null) {
```



```

    await this.apolloQLHandler.increaseReferralFee(
        ...
    );
}
}

```

## Issue

WyvernExchange에서 Referral Fee는 `sell.feeRecipient` 또는 `buy.feeRecipient`로 이동합니다. 이 값은 사용자가 컨트롤 할 수 있는 값이므로, 이 값을 사용자의 주소로 두면 Referral Fee를 사실상 내지 않으면서 동시에 백엔드에서는 Referral Fee가 증가한 상태로 만들 수 있습니다. 또한, WyvernExchange 상에서는 각종 fee rate을 전부 사용자가 지정할 수 있는데, BackEnd 상에서는 fee rate을 단순히 DB에서 가져와서 Referral Fee를 계산합니다.

현재 백엔드 로직에서는 이러한 공격을 막기 위해서 NFT가 실제로 Marketplace에 등록된 것인지, `buy hash` 또는 `sell hash`가 백엔드에 저장되어 있는 것인지를 확인하지만, 이는 공격을 막기에 충분하지 않습니다.

정상적으로 BIFROST NFT Marketplace에 등록된 NFT 하나를 갖고 있는 공격자 A가 있다 합시다. A는 두 개의 블록체인 주소 X, Y를 갖고 있으며, 주소 X가 NFT를 보유하고 있다고 합시다. 또한, 주소 X와 주소 Y는 A가 보유한 다른 주소들과 Referral을 주고 받은 관계라고 가정합니다.

1. A는 우선 주소 X에서 해당 NFT를 매우 비싼 고정가로 판매 등록을 합니다.
2. A는 주소 Y로, 100 ETH에 해당 NFT를 구매하겠다고 제안을 합니다. 이 구매 제안은 서명이 되며, 백엔드 상에 해시와 함께 저장됩니다. 공격자는 이제 해시를 생성하기 위해 만들어진 salt 등 각종 파라미터들을 확인하기 위해서 해당 제안을 취소하는 버튼을 누른 후 `cancelOrder()`의 트랜잭션 데이터를 확인한 후, abi-decoding을 하고, 트랜잭션을 거부합니다. 공격자가 백엔드에 직접 접근하는 대신 우회적으로 데이터를 확인하는 겁니다. 이때, 주소 Y가 제안한 buy order는 `feeRecipient`가 `address(0)`인 주문입니다.
3. 이제 다시 주소 X로 BIFROST NFT Marketplace의 프론트엔드를 거치지 않고 직접 주소 Y의 제안을 받아들입니다. 이때, sell order에서 모든 Fee를 0%로 설정해서 거래를 합니다. 또한, `feeRecipient`도 원하는 주소로 설정할 수 있습니다.
4. 100 ETH는 다시 A가 갖고 있는 주소(X)로 돌아오고, Referral Fee는 각각 100 ETH의 거래가 일어났을 때 발생하는 Referral Fee 만큼 A가 갖는 주소들에게 주어집니다. 즉, A는 순수하게 Referral Fee만 가져가게 됩니다.
5. 중앙화된 방식으로 Referral Fee를 분배하는 과정에서 이 사실을 눈치채지 못한다면, 실제로 A에게 자산이 탈취되어 공격이 마무리됩니다.

## Recommendation

Referral Fee에 대한 전반적인 로직 수정을 진행해주세요.

우선 컨트랙트에서 feeRecipient 주소가 실제로 referral fee들이 모이는 곳인지 확인하는 과정을 추가합니다. referral fee가 모이는 주소는 컨트랙트의 owner만이 설정할 수 있도록 합니다.

fee rate에 대한 체크는 두 가지 방법으로 가능합니다. 먼저 각 NFT 당 referral fee rate을 컨트랙트에 전부 올린 후, 온체인에서 fee rate이 제대로 입력되었는지 확인하는 방법이 있습니다.

또 다른 방법은 referral fee가 모이는 주소로 실제로 이동한 토큰의 양을 event로 백엔드에 직접 전하는 방법이 있습니다. 이러면 fee rate를 변경하더라도 프로토콜에 손해를 끼치지 않습니다.

## Update

BIFROST 측은 Exchange 주소로 보낸 트랜잭션만 고려하고, 그 중 **feeRecipient**가 정확한지, DB에서 저장한 것과 fee rate이 동일한지를 추가적으로 검사한 후 백엔드에 있는 referral fee 총량을 업데이트 하는 것으로 코드를 수정하였습니다. 또한, DB에 Order를 추가하기 전에도 feeRecipient의 정확성과 fee rate의 정확성을 확인하는 로직을 추가하였습니다.

BIFROST 측은 아래 코드에서 functionData가 event를 발생시킨 transaction data이며, 이때 transaction의 **to**가 Exchange 주소와 동일한지 확인한 뒤에 호출된다고 전달했습니다.

```
private async processWyvernExchangeOrdersMatched(
  logs: IDecodedParameterLogs[],
  transferLogs: IDecodedParameterLogs[],
  nftContract: INFTContract,
  commonData: IApplicationEventInfoData,
  functionData: IApplicationEventInfoAction
): Promise<void> {
  ...

  const abstractedOrdersMatchFunctionParameters =
    this.abstractOrdersMatchFunctionParameters(functionData.decodedParameterLogs);
  let creatorFee = 0;
  let marketFee = 0;
  let communityFee = 0;

  if (abstractedOrdersMatchFunctionParameters.feeRecipient === FEE_RECIPIENT) { //
    validate Fee recipient
    const relayerFee = collection.creatorFee + collection.serviceFee;
    const functionRelayerFee = Math.max(
      abstractedOrdersMatchFunctionParameters.makerRelayerFee,
      abstractedOrdersMatchFunctionParameters.takerRelayerFee
    );
```

```

    if (relayerFee === functionRelayerFee) { // validate service fee, creator fee,
referral fee
        // creatorFee, marketFee has decimal
        const PERCENT_NUM = 10000;
        creatorFee = Math.floor(tradePrice * collection.creatorFee / PERCENT_NUM);
        marketFee = Math.floor(tradePrice * collection.serviceFee / PERCENT_NUM);
    }

    ...

    const functionCommunityFee = Math.max(
        abstractedOrdersMatchFunctionParameters.makerProtocolFee,
        abstractedOrdersMatchFunctionParameters.takerProtocolFee
    );

    if (collection.communityFee === functionCommunityFee) {
        communityFee = Math.floor(tradePrice * collection.communityFee / nftCount);
    }
}

...
...

// inc creator fee
await this.apolloQLHandler.increaseCreatorFee(
    ...
);

// inc sender recommender referral fee
const senderReferral = await this.apolloQLHandler.findOneReferral(sender);
if (senderReferral !== null) {
    await this.apolloQLHandler.increaseReferralFee(
        ...
    );
}

// inc buyer recommender referral fee
const buyerReferral = await this.apolloQLHandler.findOneReferral(receiver);
if (buyerReferral !== null) {
    await this.apolloQLHandler.increaseReferralFee(
        ...
    );
}
}

private abstractOrdersMatchFunctionParameters(logs: IDecodedParameterLogs[]): {
    feeRecipient: string,
    makerRelayerFee: number,
    takerRelayerFee: number,
    makerProtocolFee: number,
    takerProtocolFee: number
} {
    let buyFeeRecipient: string = '';

```

```

let sellFeeRecipient: string = '';
let feeRecipient: string = '';
let makerRelayerFee: number = 0;
let takerRelayerFee: number = 0;
let makerProtocolFee: number = 0;
let takerProtocolFee: number = 0;

for (const log of logs) {
  switch (log.name) {
    case 'addrs':
      const stringAddrs = String(log.value);
      const addrs = stringAddrs.split(',');

      buyFeeRecipient = addrs[3];
      sellFeeRecipient = addrs[10];

      buyFeeRecipient !== ZERO_ADDRESS ? feeRecipient = buyFeeRecipient : feeRecipient
= sellFeeRecipient;
    case 'uints':
      const stringUints = String(log.value);
      const uints = stringUints.split(',');

      if (sellFeeRecipient !== ZERO_ADDRESS) {
        // seller is maker case
        const sellMakerRelayerFee = uints[9];
        const sellTakerRelayerFee = uints[10];
        const sellMakerProtocolFee = uints[11];
        const sellTakerProtocolFee = uints[12];

        makerRelayerFee = +sellMakerRelayerFee;
        takerRelayerFee = +sellTakerRelayerFee;
        makerProtocolFee = +sellMakerProtocolFee;
        takerProtocolFee = +sellTakerProtocolFee;
      } else {
        // buyer is maker case
        const buyMakerRelayerFee = uints[0];
        const buyTakerRelayerFee = uints[1];
        const buyMakerProtocolFee = uints[2];
        const buyTakerProtocolFee = uints[3];

        makerRelayerFee = +buyMakerRelayerFee;
        takerRelayerFee = +buyTakerRelayerFee;
        makerProtocolFee = +buyMakerProtocolFee;
        takerProtocolFee = +buyTakerProtocolFee;
      }
    }
  }
  return { feeRecipient, makerRelayerFee, takerRelayerFee, makerProtocolFee,
takerProtocolFee };
}

```

## ⚠ MAJOR [Contract] Wyvern v2.2의 Denial of Service Vulnerability가 패치되지 않았습니다 (Resolved - v.1.0)

```
contract WyvernProxyRegistry is ProxyRegistry {

    string public constant name = "Project Wyvern Proxy Registry";

    /* Whether the initial auth address has been set. */
    bool public initialAddressSet = false;

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

    /**
     * Grant authentication to the initial Exchange protocol contract
     *
     * @dev No delay, can only be called once - after that the standard registry process with a
     * delay must be used
     * @param authAddress Address of the contract to grant authentication
     */
    function grantInitialAuthentication (address authAddress)
        onlyOwner
        public
    {
        require(!initialAddressSet);
        initialAddressSet = true;
        contracts[authAddress] = true;
    }

}
```

### Issue

Wyvern v2.2에서 발견되었던 Denial of Service 취약점이 패치가 되지 않았습니다.

취약점에 대해서는 이미 작성된 Post Mortem 자료가 있으므로, 이를 참고하시기 바랍니다.

<https://medium.com/wyvern-protocol/post-mortem-wyvern-v2-2-denial-of-service-vulnerability-bdc528af897e>

이 취약점은 Wyvern v3에서 패치되었습니다.

<https://github.com/wyvernprotocol/wyvern-v3/commit/8d8cf1cfaac723710043448f36e5a007d159e4ec>

### Update

BIFROST 측은 이를 확인하고, Wyvern v3와 동일한 패치를 적용했습니다.

○ **MINOR** [Contract] AuctionMatcher의 transferOwnership의 require 문이 잘못되었습니다. (Resolved - v.2.0)

```
function transferOwnership(address _newOwner) public onlyOwner {  
    require(_newOwner == address(0));  
  
    emit OwnershipTransferred(owner, _newOwner);  
    owner = _newOwner;  
}
```

### Issue

require 문의 `==` 이 `!=` 으로 바뀌어야 합니다.

### Recommendation

require 문의 `==` 이 `!=` 으로 바꾸는 것을 제안합니다. 이와 별개로 현재 Owner 권한과 관련된 Contract 및 그 구현 방식이 다양한데, 이를 하나의 Contract로 통일하는 것을 추천합니다.

### Update

BIFROST 측은 해당 require 문을 수정했다고 전달했습니다.

**MINOR** [Contract] ERC20 Transfer의 return 값을 확인하지 않고 있습니다. (Found - v.2.0)

```
if(rewardAmount[tokenAddrIndex] > 0){  
    if(tokenAddrs[tokenAddrIndex] == NATIVECOINADDR){  
        payable(msg.sender).transfer(rewardAmount[tokenAddrIndex]);  
    }else{  
        IERC20 token = IERC20(tokenAddrs[tokenAddrIndex]);  
        uint256 underlyingDecimal = uint256(10 ** token.decimals());  
        uint256 underlyingAmount =  
rewardAmount[tokenAddrIndex].unifiedToUnderlyingAmount(underlyingDecimal);  
        token.transfer(msg.sender, underlyingAmount);  
    }  
}
```

### Issue

ERC20 transfer의 return 값이 있다면, 이 값이 true인지 확인해야 합니다.

### Recommendation

SafeERC20 등 방법으로 transfer 문의 return 값이 있다면 true인지 확인하는 것을 추천합니다.

### Disclaimer

BIFROST NFT Marketplace에서 해당 이슈가 발생하기 위해서는, NFT Marketplace에서 지원하는 ERC20가 전송에 실패했을때 revert 하지 않고 false를 반환해야 합니다. 하지만 보안감사 시점의 BIFROST NFT Marketplace에서 지원하는 결제 수단은 해당 이슈가 발생하지 않는 것으로 확인되었습니다. 따라서 추가적인 결제 수단을 지원하지 않는 경우 위 이슈는 발생하지 않을 것으로 보여 이미 배포된 경우 재배포를 요하는 상황은 아닙니다. 하지만 추후 의사 결정 과정에서 추가될 수 있으므로 아직 배포되지 않았다면 위 Recommendation 대로 추가하시는 것을 추천드리고, 이미 배포하여 운용 중이라면 의사 결정에 참고하시길 바랍니다.

## ○ MINOR

[Out-of-Scope] ImgProxy에서 임의의 서버를 대상으로 리퀘스트를 발생시킬 수 있습니다.

(Resolved - v.2.0)

### Issue

/api/signUrl 엔드 포인트에서 <https://d388oretzkmm3w.cloudfront.net> 를 이용한 이미지 링크를 생성하는 것을 확인했습니다. <https://d388oretzkmm3w.cloudfront.net> 의 경우 이번 Audit의 대상은 아니지만, Informational 한 정보라 알고 계시면 좋을 것 같아서 추가하였습니다. 해당 주소에서는 ImgProxy 3.5를 쓰고 계신것으로 확인되며, 이미지를 파싱해서 저장하고 리사이징이나 컨버팅이 가능한 것으로 확인되었습니다. 문제는 파싱하는 대상이 화이트리스트 방식으로 운영되지 않아서, ImgProxy 내부의 API에 접근하거나 외부 서버에 GET Method를 이용한 리퀘스트를 발생시킬 수 있다는 점 입니다. 추가로 공식 document ([https://docs.imgproxy.net/3.5/image\\_formats\\_support](https://docs.imgproxy.net/3.5/image_formats_support)) 를 참고하면, 이미지 외에도 pdf나 비디오 파일을 업로드할 수 있는 것을 알 수 있습니다. pdf를 이용할 경우, 특정 공격이 가능할 수도 있습니다.

```
{"url":"https://d388oretzkmm3w.cloudfront.net/wmBE6NGGyNmv53uX-SoyXgW1QpShs6luk4sgyNkGpvE/rs:fit:600:0:1/format:webp/ahP0cDovL3Z1bG4ubG12ZTozMzMzNw"}
```

/api/signUrl에서 생성한 url

```
sqrtrev@vm:~$ nc -lnvp 31337
Listening on 0.0.0.0 31337
Connection received on 54.180.131.93 13716
GET / HTTP/1.1
Host: vuln.live:31337
User-Agent: imgproxy/3.5.0
```

임의의 서버로 request 성공적으로 돌린 화면

### Recommendation

<https://docs.imgproxy.net/configuration?id=security> 공식 Documentation의 Security를 확인하면 IMGPROXY\_ALLOWED\_SOURCES 옵션을 활용하여 화이트리스트 방식으로 전환할 수 있는 것을 확인하였습니다. 해당 기능을 이용하시기를 권장드립니다.

### Update

Recommendation에 첨부한 security configuration 을 이용하여 화이트리스트 방식으로 전환하여 이슈를 해결함을 확인했습니다.



## ◉ MINOR

[Backend] marketplace\_apollo의 api에 외부 사이트에서 리퀘스트를 보낼 수 있습니다.

(Resolved - v.2.0)

### Issue

`access-control-allow-origin` 옵션이 \*로 설정되어있습니다. CORS를 허용하기 위해서 설정해준 것으로 생각되지만 \*로 설정할 경우 의도하지 않은 도메인에서 해당 API에 xhr이나 fetch와 같은 방법을 통해서 Apollo API와 통신이 가능합니다.

### Recommendation

`access-control-allow-origin`는 신뢰할 수 있는 도메인으로 한정되어야 합니다.  
일반적으로는 API를 호출하는 도메인으로 고정하는것이 좋습니다.

### Update

`access-control-allow-origin`을 BIFROST의 페이지로 제한하여 해당 이슈를 해결하였습니다.

**TIPS** [Contract] Wyvern v2.2는 이더리움 메인넷에서 USDT를 지원하지 않습니다.  
(Resolved - v.1.0)

```
[ChainID.ETH_MAIN, [
  {
    paymentType: PaymentType.NATIVE,
    tokenAddress: "0x0000000000000000000000000000000000000000",
    decimal: 18
  },
  {
    paymentType: PaymentType.WETH,
    tokenAddress: "0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2",
    decimal: 18
  },
  {
    paymentType: PaymentType.BFC,
    tokenAddress: "0x0c7d5ae016f806603cb1782bea29ac69471cab9c",
    decimal: 18
  },
  {
    paymentType: PaymentType.USDT,
    tokenAddress: "0xdac17f958d2ee523a2206206994597c13d831ec7",
    decimal: 6
  },
  {
    paymentType: PaymentType.USDC,
    tokenAddress: "0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48",
    decimal: 6
  }
]
```

## Issue

Wyvern v2.2에서는 Token을 Transfer 한다음 그 return 값이 true인지를 확인하기 때문에, ERC20의 스펙을 정확하게 만족시키지 않는 자산을 지원할 수 없습니다. 이러한 자산 중 하나는 approve, transfer, transferFrom에서 bool을 return 하지 않는 ETH 메인넷의 USDT입니다.

## Recommendation

ETH 메인넷의 USDT를 지원하는 자산 목록에서 제외하는 것을 추천합니다.  
별개로 다른 자산을 지원하는 것을 고려할 때도 위 사실을 추가로 검토하는 것을 추천합니다.

## Update

BIFROST 팀에서 ETH 메인넷의 USDT를 지원하지 않는 것으로 이슈를 해결하였습니다.

**💡 TIPS** [Contract] Solidity Version을 고정하는 것을 추천합니다. (Resolved - v.2.0)

### Issue

Solidity Version을 ^0.8.13으로 두는 것보다 0.8.13과 같이 고정하는 것이 더 안전합니다.

### Recommendation

다만, Solidity 0.8.13은 버그가 발견되어, 최신 버전인 0.8.15로 고정하는 것을 제안합니다.

<https://blog.soliditylang.org/2022/06/15/inline-assembly-memory-side-effects-bug/>

### Update

BIFROST 측은 모든 Solidity 버전을 0.8.15로 고정했음을 전달했습니다.

# DISCLAIMER

---

해당 리포트는 투자에 대한 조언, 비즈니스 모델의 적합성, 버그 없이 안전한 코드를 보증하지 않습니다. 해당 리포트는 알려진 기술 문제들에 대한 논의의 목적으로만 사용됩니다. 리포트에 기술된 문제 외에도 메인넷 상의 결함 등 발견되지 않은 문제들이 있을 수 있습니다. 안전한 스마트 컨트랙트를 작성하기 위해서는 발견된 문제들에 대한 수정과 충분한 테스트가 필요합니다.

---

**End of Document**