2022-06-infinity (Infinity NFT Marketplace)

Benchmark Security Review Performed by Bug Hunter

23.06.2025

Table of Contents

About Bug Hunter

Bug Hunter is an automated code-review tool for Solidity smart contracts, developed by Truscova GmbH. It uses proprietary algorithms and advanced code-analysis techniques to identify potential vulnerabilities efficiently.

Bug Hunter is designed to support developers in the Ethereum ecosystem by delivering consistent, scalable assessments that align with best practices in Web 3.0 security. Reports are generated via the Bug Hunter tool and are accessible to clients through a secure dashboard web interface or relevant Git repositories.

Bug Hunter is a product of Truscova, a security R&D company headquartered in Bremen, Germany. Founded in late 2022, Truscova specializes in formal verification, fuzzing, dynamic and static analysis, and security tooling. Its founding team includes researchers with 35+ books, several patents, and over 20 000 citations in software testing, verification, security, and machine learning.

To explore more about Bug Hunter, visit:

- Bug Hunter Website
- Bug Hunter Documentation
- Example Reports

To explore more about Truscova, visit:

- Truscova Website
- X
- LinkedIn

Notices and Remarks

Copyright and Distribution

This security-review report was generated with Truscova's Bug Hunter tool to benchmark its performance on selected NFT projects. It is made available through the Bug Hunter dashboard and relevant Git repositories. Users may choose to share or publish this report at their own discretion.

Test Coverage Disclaimer

This security review was conducted exclusively by Bug Hunter, an automated code-review tool for Solidity codebases. Bug Hunter uses a combination of proprietary algorithms and programmatic techniques to identify potential vulnerabilities in Solidity smart contracts.

The findings in this report should not be considered a complete or exhaustive list of all possible security issues in the reviewed codebase and are dependent on Bug Hunter's coverage of detector rules. The full list of rules covered by Bug Hunter automated code review is available at https://docs.bughunter.live/.

Executive Summary

Overview and Scope

A codebase from project 2022-06-infinity was reviewed for security vulnerabilities by Bug Hunter with the following details:

- GitHub Repository: https://github.com/code-423n4/2022-06-infinity
  Commit hash: `765376fa238bbccd8b1e2e12897c91098c7e5ac6`

Bug Hunter reviewed the following files:

InfinityOrderBookComplication.sol InfinityExchange.sol InfinityStaker.sol

Summary of Findings

The uncovered vulnerabilities identified during the security review are summarised in the table below:

Severity	Count
High	6
Medium	6
Low	3

Project Summary

Contact Information

For support, contact us at support@bughunter.live

Project Timeline

Date	Event
23.06.2025	Project submitted
23.06.2025	Security review concluded

Detailed Findings

1. Missing Validation of Distinct Token IDs in doItemsIntersect() for Order Matching

Severity	High
Finding ID	BH-2022-06-infinity-01
Target	contracts/core/InfinityOrderBookComplication.sol
Function name	doItemsIntersect

Description

The function `doItemsIntersect` checks that collections match between maker and taker orders and that token IDs intersect within those collections. However, it does not validate that the actual token IDs used by the taker match the exact ones specified by the maker in the signed order. This is especially dangerous for ERC1155 tokens where multiple instances of the same token ID exist. A taker can exploit this by transferring different token IDs or repeating token IDs across collections, which may still pass the intersection checks. As a result, users may receive tokens they did not agree to or lose intended assets, violating the trust and safety guarantees of the matching engine.

```
function canExecTakeOrder(OrderTypes.MakerOrder calldata makerOrder,
OrderTypes.OrderItem[] calldata takerItems)
    external
    view
    override
    returns (bool)
  {
    return (makerOrder.constraints[3] <= block.timestamp &&
      makerOrder.constraints[4] >= block.timestamp &&
      areTakerNumItemsValid(makerOrder, takerItems) &&
      doItemsIntersect(makerOrder.nfts, takerItems));
  }

function areTakerNumItemsValid(OrderTypes.MakerOrder calldata makerOrder,
OrderTypes.OrderItem[] calldata takerItems)
    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;
  }

function doItemsIntersect(OrderTypes.OrderItem[] calldata order1Nfts,
OrderTypes.OrderItem[] calldata order2Nfts)
    public
    pure
    returns (bool)
  {
    uint256 order1NftsLength = order1Nfts.length;
    uint256 order2NftsLength = order2Nfts.length;
    // case where maker/taker didn't specify any items
    if (order1NftsLength == 0 || order2NftsLength == 0) {
      return true;
    }

    uint256 numCollsMatched = 0;
    // check if taker has all items in maker
    for (uint256 i = 0; i < order2NftsLength; ) {
      for (uint256 j = 0; j < order1NftsLength; ) {
        if (order1Nfts[j].collection == order2Nfts[i].collection) {
          // increment numCollsMatched
          unchecked {
            ++numCollsMatched;
          }
          // check if tokenIds intersect
          bool tokenIdsIntersect = doTokenIdsIntersect(order1Nfts[j],
order2Nfts[i]);
          require(tokenIdsIntersect, 'tokenIds dont intersect');
          // short circuit
          break;
        }
        unchecked {
          ++j;
```

```
        }
      }
      unchecked {
        ++i;
      }
    }

    return numCollsMatched == order2NftsLength;
  }
```

Recommendation

Enhance `doItemsIntersect` to enforce that each token ID in the takerItems exactly matches one of the token IDs specified in the makerOrder. For collections that allow duplicate token IDs (e.g., ERC1155), ensure that both the token ID and amount match. Additionally, consider validating the uniqueness of token IDs across collections if the order model assumes distinct tokens.

2. Missing Revert for Unsupported Token Standards in _transferNFTs()

Severity	High
Finding ID	BH-2022-06-infinity-02
Target	contracts/core/InfinityExchange.sol
Function name	_transferNFTs

Description

The `_transferNFTs` function silently does nothing if the given collection address does not support either ERC721 or ERC1155 interfaces. This may lead to unexpected execution paths where transfers fail silently but the transaction proceeds, causing asset loss or inconsistent state. The lack of a fallback check also makes the function vulnerable to malicious or misconfigured contracts.

```
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(0xd9b67a26)) {
      _transferERC1155s(from, to, item);
    }
  }
```

Recommendation

Add an explicit revert statement in the `else` clause to catch unsupported token standards. For example: `revert("Unsupported token standard")`. Optionally, consider interface checks during order validation to fail fast.

3. Incorrect Interface Priority in _transferNFTs() Causes Under-Transfer for Hybrid Tokens

Severity	High
Finding ID	BH-2022-06-infinity-03
Target	contracts/core/InfinityExchange.sol
Function name	_transferNFTs

Description

The `_transferNFTs` function checks for ERC721 support before ERC1155. Some NFT contracts support both interfaces. When this occurs, the function selects the ERC721 transfer path, even when the token behaves as an ERC1155 (multi-copy) token. This leads to only one token being transferred, even if the user ordered multiple. As a result, the user may receive fewer tokens than expected, breaking execution integrity and potentially causing asset loss.

```
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(0xd9b67a26)) {
      _transferERC1155s(from, to, item);
    }
  }
```

Recommendation

Reverse the interface check order: prioritize ERC1155 before ERC721. This ensures that tokens capable of multi-quantity transfers are handled correctly. Also, consider adding validation for quantity expectations during order validation.

4. Missing Refund for ETH Overpayment in takeOrders()

Severity	High
Finding ID	BH-2022-06-infinity-04
Target	contracts/core/InfinityExchange.sol
Function name	takeOrders

Description

The `takeOrders` function collects ETH payments from the caller via `msg.value` but only checks that it is greater than or equal to the total execution price. It does not refund any excess ETH back to the sender. This can result in ETH being unintentionally locked in the contract if the user overpays, especially when interacting via interfaces or scripts that estimate loosely.

```
function takeOrders(OrderTypes.MakerOrder[] calldata makerOrders,
OrderTypes.OrderItem[][] calldata takerNfts)
    external
    payable
    nonReentrant
  {
    uint256 ordersLength = makerOrders.length;
    require(ordersLength == takerNfts.length, 'mismatched lengths');
    uint256 totalPrice;
    address currency = makerOrders[0].execParams[1];
    bool isMakerSeller = makerOrders[0].isSellOrder;
    if (!isMakerSeller) {
      require(currency != address(0), 'offers only in ERC20');
    }
    for (uint256 i = 0; i < ordersLength; ) {
      require(currency == makerOrders[i].execParams[1], 'cannot mix currencies');
      require(isMakerSeller == makerOrders[i].isSellOrder, 'cannot mix order
sides');
      uint256 execPrice = _getCurrentPrice(makerOrders[i]);
      totalPrice += execPrice;
      _takeOrders(makerOrders[i], takerNfts[i], execPrice);
      unchecked {
        ++i;
      }
    }
```

```
    if (isMakerSeller && currency == address(0)) {
      require(msg.value >= totalPrice, 'invalid total price');
    }
  }
```

Recommendation

Add a refund statement after verifying `msg.value >= totalPrice`, such as `if (msg.value > totalPrice) { payable(msg.sender).transfer(msg.value - totalPrice); }` to return any excess ETH to the buyer.

5. Unsafe Balance Accounting Due to Snapshot Dependency in unstake() and _updateUserStakedAmounts()

Severity	High
Finding ID	BH-2022-06-infinity-05
Target	contracts/core/InfinityExchange.sol
Function name	unstake

Description

The `unstake` function calculates the user's total available balance by summing staking amounts across all durations and passes those pre-read values into `_updateUserStakedAmounts` to perform deductions. However, this update logic assumes that the passed-in snapshot values remain accurate at the time of deduction. If the staking state is modified in between (e.g., via reentrancy or concurrent calls), this can lead to incorrect deductions, inconsistencies, or even over-withdrawals. The nested if-else structure further complicates control flow and increases the chance of accounting errors.

```
function unstake(uint256 amount) external override nonReentrant whenNotPaused {
    require(amount != 0, 'stake amount cant be 0');
    uint256 noVesting = userstakedAmounts[msg.sender][Duration.NONE].amount;
    uint256 vestedThreeMonths = getVestedAmount(msg.sender, Duration.THREE_MONTHS);
    uint256 vestedsixMonths = getVestedAmount(msg.sender, Duration.SIX_MONTHS);
    uint256 vestedTwelveMonths = getVestedAmount(msg.sender,
Duration.TWELVE_MONTHS);
    uint256 totalVested = noVesting + vestedThreeMonths + vestedsixMonths +
vestedTwelveMonths;
    require(totalVested >= amount, 'insufficient balance to unstake');

    _updateUserStakedAmounts(msg.sender, amount, noVesting, vestedThreeMonths,
vestedsixMonths, vestedTwelveMonths);
    IERC20(INFINITY_TOKEN).safeTransfer(msg.sender, amount);
    emit UnStaked(msg.sender, amount);
  }
```

Recommendation

Refactor `_updateUserStakedAmounts` to read the current staking values from storage directly instead of relying on passed-in snapshot values. Implement a sequential or loop-based approach that deducts the `amount` across durations and ensure the final deducted amount equals the requested amount. Consider adding state integrity assertions to enforce accounting correctness.

6. areNumItemsValid() Allows Over-Delivery to Buyer, Violating Quantity Constraints

Severity	High
Finding ID	BH-2022-06-infinity-06
Target	contracts/core/InfinityOrderBookComplication.sol
Function name	areNumItemsValid

Description

The `areNumItemsValid` function checks that the constructed item count is at least what the buyer requested (`buy.constraints[0]`) and that the buyer's constraint is within the seller's offer. However, it does not prevent the actual transfer count (`numConstructedItems`) from exceeding the buyer's constraint. This can lead to over-delivery of tokens, violating the buyer's intent and potentially causing unexpected costs or state inconsistencies if token amount is price-sensitive.

```
function areNumItemsValid(
    OrderTypes.MakerOrder calldata sell,
    OrderTypes.MakerOrder calldata buy,
    OrderTypes.OrderItem[] calldata constructedNfts
  ) public pure returns (bool) {
    uint256 numConstructedItems = 0;
    uint256 nftsLength = constructedNfts.length;
    for (uint256 i = 0; i < nftsLength; ) {
      unchecked {
        numConstructedItems += constructedNfts[i].tokens.length;
        ++i;
      }
    }
    return numConstructedItems >= buy.constraints[0] && buy.constraints[0] <=
sell.constraints[0];
  }
```

Recommendation

Modify the logic to ensure `numConstructedItems == buy.constraints[0]`, or explicitly cap the transferred items at the buyer's requested quantity elsewhere in the code. Only allow over-delivery if the protocol explicitly supports it.

7. Gas Cost Manipulation and Lack of Bounds Check in Gas Refund Mechanism

Severity	Medium
Finding ID	BH-2022-06-infinity-07
Target	contracts/core/InfinityExchange.sol
Function name	_execMatchOrders

Description

The function `_execMatchOrders` calculates the gas cost incurred during execution using the formula `(startGasPerOrder - gasleft() + wethTransferGasUnits) * tx.gasprice`. This value is then reimbursed by transferring the calculated amount of WETH (or other tokens) from the buyer to the contract. However, there is no upper bound or sanity check on the gas cost calculated, meaning a malicious seller could artificially increase the gas usage to inflate this reimbursement. This creates a griefing vector where the buyer ends up paying excessive gas cost refunds. Additionally, if `startGasPerOrder < gasleft()`, this could result in underflows (though in Solidity >=0.8.0 this would revert), but the intent still needs to be validated.

```
function _execMatchOrders(
    bytes32 sellOrderHash,
    bytes32 buyOrderHash,
    OrderTypes.MakerOrder calldata sell,
    OrderTypes.MakerOrder calldata buy,
    OrderTypes.OrderItem[] calldata constructedNfts,
    uint256 startGasPerOrder,
    uint256 execPrice,
    uint16 protocolFeeBps,
    uint32 wethTransferGasUnits,
    address weth
  ) internal {
    uint256 protocolFee = (protocolFeeBps * execPrice) / 10000;
    uint256 remainingAmount = execPrice - protocolFee;
    _execMatchOrder(
      sell.signer,
      buy.signer,
      sell.constraints[5],
      buy.constraints[5],
```

```
      constructedNfts,
      buy.execParams[1],
      remainingAmount
    );
    _emitMatchEvent(
      sellOrderHash,
      buyOrderHash,
      sell.signer,
      buy.signer,
      buy.execParams[0],
      buy.execParams[1],
      execPrice
    );
    uint256 gasCost = (startGasPerOrder - gasleft() + wethTransferGasUnits) *
tx.gasprice;
    // if the execution currency is weth, we can send the protocol fee and gas cost
in one transfer to save gas
    // else we need to send the protocol fee separately in the execution currency
    if (buy.execParams[1] == weth) {
      IERC20(weth).safeTransferFrom(buy.signer, address(this), protocolFee +
gasCost);
    } else {
      IERC20(buy.execParams[1]).safeTransferFrom(buy.signer, address(this),
protocolFee);
      IERC20(weth).safeTransferFrom(buy.signer, address(this), gasCost);
    }
  }
```

Recommendation

Introduce a cap or maximum allowed gas cost that can be charged to the buyer. Also, validate that `startGasPerOrder >= gasleft()` before performing subtraction. Consider including buyer-signed agreement on gas cost bounds off-chain or within the order structure itself.

8. Unclaimed ETH Sent for Non-ETH Orders May Become Stuck in Contract

Severity	Medium
Finding ID	BH-2022-06-infinity-08
Target	contracts/core/InfinityExchange.sol
Function name	takeMultipleOneOrders

Description

In the `takeMultipleOneOrders` function, ETH is only validated via `require(msg.value >= totalPrice)` if the maker is a seller and the currency is ETH (`address(0)`). However, if the order is in ERC20 (i.e., not native ETH), and the caller mistakenly sends ETH with the transaction (i.e., `msg.value > 0`), there is no mechanism to refund or reject it. This could result in ETH being permanently stuck in the contract unless there is a withdrawal mechanism implemented elsewhere. This behavior may confuse users and lead to unintentional fund loss.

```
function takeMultipleOneOrders(OrderTypes.MakerOrder[] calldata makerOrders)
external payable nonReentrant {
    uint256 numMakerOrders = makerOrders.length;
    uint256 totalPrice;
    address currency = makerOrders[0].execParams[1];
    bool isMakerSeller = makerOrders[0].isSellOrder;
    if (!isMakerSeller) {
      require(currency != address(0), 'offers only in ERC20');
    }
    for (uint256 i = 0; i < numMakerOrders; ) {
      bytes32 makerOrderHash = _hash(makerOrders[i]);
      require(isOrderValid(makerOrders[i], makerOrderHash), 'invalid maker order');
      bool isTimeValid = makerOrders[i].constraints[3] <= block.timestamp &&
        makerOrders[i].constraints[4] >= block.timestamp;
      require(isTimeValid, 'invalid time');
      require(currency == makerOrders[i].execParams[1], 'cannot mix currencies');
      require(isMakerSeller == makerOrders[i].isSellOrder, 'cannot mix order
sides');
      uint256 execPrice = _getCurrentPrice(makerOrders[i]);
      totalPrice += execPrice;
      _execTakeOneOrder(makerOrderHash, makerOrders[i], isMakerSeller, execPrice);
      unchecked {
```

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

Recommendation

Add a check at the beginning of the function to reject ETH sent for non-ETH orders. For example:

```
require(currency == address(0) || msg.value == 0, 'ETH not accepted for ERC20
orders');
```

Alternatively, provide a mechanism for users to reclaim mistaken ETH transfers through a withdrawal function.

9. Potential Token Theft via Empty NFT Orders in matchOneToManyOrders

Severity	Medium
Finding ID	BH-2022-06-infinity-09
Target	contracts/core/InfinityExchange.sol
Function name	matchOneToManyOrders

Description

The function `matchOneToManyOrders` processes one maker order against multiple counterparty orders. However, it lacks validation that the `manyMakerOrders` array contains orders that actually transfer meaningful NFTs. If an attacker fills the array with orders that are valid but transfer zero NFTs (e.g., metadata manipulation or empty collections), the maker (buyer or seller) could still be charged the full `execPrice`, `protocolFee`, and `gasCost`. In buy-side execution, this could lead to the buyer paying for nothing and having tokens stolen through protocol fees and gas reimbursements, particularly if the `canExecMatchOneToMany` check is improperly implemented or lenient.

```
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]), 'invalid
complication');
    require(
      IComplication(makerOrder.execParams[0]).canExecMatchOneToMany(makerOrder,
manyMakerOrders),
      'cannot execute'
    );
    bytes32 makerOrderHash = _hash(makerOrder);
    require(isOrderValid(makerOrder, makerOrderHash), 'invalid maker order');
    uint256 ordersLength = manyMakerOrders.length;
    uint16 protocolFeeBps = PROTOCOL_FEE_BPS;
    uint32 wethTransferGasUnits = WETH_TRANSFER_GAS_UNITS;
    address weth = WETH;
```

```
    if (makerOrder.isSellOrder) {
      for (uint256 i = 0; i < ordersLength; ) {
        uint256 startGasPerOrder = gasleft() + ((startGas + 20000 - gasleft()) /
ordersLength);
        _matchOneMakerSellToManyMakerBuys(
          makerOrderHash,
          makerOrder,
          manyMakerOrders[i],
          startGasPerOrder,
          protocolFeeBps,
          wethTransferGasUnits,
          weth
        );
        unchecked {
          ++i;
        }
      }
      isUserOrderNonceExecutedOrCancelled[makerOrder.signer]
[makerOrder.constraints[5]] = true;
    } else {
      uint256 protocolFee;
      for (uint256 i = 0; i < ordersLength; ) {
        protocolFee += _matchOneMakerBuyToManyMakerSells(
          makerOrderHash,
          manyMakerOrders[i],
          makerOrder,
          protocolFeeBps
        );
        unchecked {
          ++i;
        }
      }
      isUserOrderNonceExecutedOrCancelled[makerOrder.signer]
[makerOrder.constraints[5]] = true;
      uint256 gasCost = (startGas - gasleft() + WETH_TRANSFER_GAS_UNITS) *
tx.gasprice;
      if (makerOrder.execParams[1] == weth) {
        IERC20(weth).safeTransferFrom(makerOrder.signer, address(this), protocolFee
+ gasCost);
      } else {
        IERC20(makerOrder.execParams[1]).safeTransferFrom(makerOrder.signer,
address(this), protocolFee);
        IERC20(weth).safeTransferFrom(makerOrder.signer, address(this), gasCost);
      }
```

```
    }
  }
```

Recommendation

Ensure each order in `manyMakerOrders` contains valid and non-empty NFTs before executing. Validate in `canExecMatchOneToMany` or directly within the loop:

```
require(_hasValidNFTs(manyMakerOrders[i]), 'empty or invalid NFT');
```

Also, review the implementation of `_matchOneMakerBuyToManyMakerSells` and `_matchOneMakerSellToManyMakerBuys` to confirm that no value-less transfers are allowed.

10. Unbounded Gas Refund Value Allows Owner to Steal User Funds

Severity	Medium
Finding ID	BH-2022-06-infinity-10
Target	contracts/core/InfinityExchange.sol
Function name	updateWethTranferGas

Description

The function `updateWethTranferGas` allows the contract owner to arbitrarily set the `WETH_TRANSFER_GAS_UNITS` value, which is directly used in gas reimbursement calculations throughout the marketplace (e.g., in `matchOneToOneOrders`, `matchOneToManyOrders`, etc.). If the owner sets this to an excessively high value, it inflates the gas reimbursement amount, resulting in buyers unknowingly overpaying WETH during order execution. This creates a backdoor where the owner can siphon funds from users under the pretext of 'gas reimbursement'.

```
function updateWethTranferGas(uint32 _wethTransferGasUnits) external onlyOwner {
    WETH_TRANSFER_GAS_UNITS = _wethTransferGasUnits;
    emit NewWethTransferGasUnits(_wethTransferGasUnits);
  }
```

Recommendation

Impose a strict upper bound on `_wethTransferGasUnits` (e.g., 21,000 to 100,000) to prevent abuse:

```
require(_wethTransferGasUnits <= MAX_REASONABLE_GAS_UNITS, 'value too high');
```

Also, clearly define and document acceptable limits for gas refunds.

11. Inadequate Validation May Allow Empty or Malicious NFT Orders to Execute

Severity	Medium
Finding ID	BH-2022-06-infinity-11
Target	contracts/core/InfinityOrderBookComplication.sol
Function name	canExecMatchOrder

Description

The function `canExecMatchOrder` is responsible for determining whether a sell and buy order can be matched. However, it lacks explicit checks to ensure that the `constructedNfts` array, or the `sell.nfts` and `buy.nfts` fields, are not empty or malformed. If these arrays contain no valid NFT items (e.g., empty arrays or placeholder entries), the function may still return `true`, allowing malicious orders that request payment but transfer nothing. This creates a potential token theft or fund-drain vector if used by a malicious actor in combination with upstream functions like `matchOneToOneOrders` or `matchOneToManyOrders`.

```
function canExecMatchOrder(
    OrderTypes.MakerOrder calldata sell,
    OrderTypes.MakerOrder calldata buy,
    OrderTypes.OrderItem[] calldata constructedNfts
  ) external view override returns (bool, uint256) {
    (bool _isPriceValid, uint256 execPrice) = isPriceValid(sell, buy);
    return (
      isTimeValid(sell, buy) &&
        _isPriceValid &&
        areNumItemsValid(sell, buy, constructedNfts) &&
        doItemsIntersect(sell.nfts, constructedNfts) &&
        doItemsIntersect(buy.nfts, constructedNfts) &&
        doItemsIntersect(sell.nfts, buy.nfts),
      execPrice
    );
  }
```

Recommendation

Add explicit validation to ensure that `constructedNfts`, `sell.nfts`, and `buy.nfts` are non-empty and contain valid token data. For example:

```
require(constructedNfts.length > 0, 'constructedNfts cannot be empty');
```

Also verify each NFT entry contains valid token addresses and IDs. This will prevent users from executing 'empty' trades.

12. Missing NFT Transfer Validation and Gas Refund Abuse in _execMatchOneToOneOrders

Severity	Medium
Finding ID	BH-2022-06-infinity-12
Target	contracts/core/InfinityExchange.sol
Function name	_execMatchOneToOneOrders

Description

The function `_execMatchOneToOneOrders` transfers NFTs and payment between buyer and seller, but it does not validate that the **`sell.nfts`** array is non-empty or that it contains valid NFTs. This opens a vulnerability where the seller sends an empty NFT list, and the buyer still transfers payment, protocol fees, and gas refunds — effectively allowing token theft. Additionally, gas cost calculation using `(startGasPerOrder - gasleft() + wethTransferGasUnits) * tx.gasprice` can be manipulated, especially without upper bounds, leading to excessive fund transfers under the guise of gas refunds.

```
function _execMatchOneToOneOrders(
    bytes32 sellOrderHash,
    bytes32 buyOrderHash,
    OrderTypes.MakerOrder calldata sell,
    OrderTypes.MakerOrder calldata buy,
    uint256 startGasPerOrder,
    uint256 execPrice,
    uint16 protocolFeeBps,
    uint32 wethTransferGasUnits,
    address weth
  ) internal {
    isUserOrderNonceExecutedOrCancelled[sell.signer][sell.constraints[5]] = true;
    isUserOrderNonceExecutedOrCancelled[buy.signer][buy.constraints[5]] = true;
    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.signer,
remainingAmount);
    _emitMatchEvent(
      sellOrderHash,
```

```
      buyOrderHash,
      sell.signer,
      buy.signer,
      buy.execParams[0],
      buy.execParams[1],
      execPrice
    );
    uint256 gasCost = (startGasPerOrder - gasleft() + wethTransferGasUnits) *
tx.gasprice;
    // if the execution currency is weth, we can send the protocol fee and gas cost
in one transfer to save gas
    // else we need to send the protocol fee separately in the execution currency
    if (buy.execParams[1] == weth) {
      IERC20(weth).safeTransferFrom(buy.signer, address(this), protocolFee +
gasCost);
    } else {
      IERC20(buy.execParams[1]).safeTransferFrom(buy.signer, address(this),
protocolFee);
      IERC20(weth).safeTransferFrom(buy.signer, address(this), gasCost);
    }
  }
```

Recommendation

1. Add validation to ensure `sell.nfts` is non-empty and contains valid token addresses and IDs:

```
require(sell.nfts.length > 0, 'no NFTs to transfer');
```

1. Cap the gas refund amount to a reasonable limit (e.g., 100,000 gas units).
2. Consider requiring the buyer to explicitly approve and agree to the gas refund mechanism off-chain or include limits in order constraints.

13. Immutable Critical Parameters Not Enforced or Validated in Constructor

Severity	Low
Finding ID	BH-2022-06-infinity-13
Target	contracts/core/InfinityExchange.sol
Function name	constructor

Description

The constructor initializes critical variables like `WETH`, `MATCH_EXECUTOR`, and `DOMAIN_SEPARATOR`, but it does not perform any validation on the `_WETH` or `_matchExecutor` addresses. If either address is set to an invalid or malicious contract (e.g., a fake WETH token that reverts or steals funds), it could compromise the entire exchange. Additionally, `WETH` and `MATCH_EXECUTOR` are not declared `immutable`, so although they are only set in the constructor, marking them `immutable` would save gas and enforce safety.

```
constructor(address _WETH, address _matchExecutor) {
    // Calculate the domain separator
    DOMAIN_SEPARATOR = keccak256(
      abi.encode(
        keccak256('EIP712Domain(string name,string version,uint256 chainId,address
verifyingContract)'),
        keccak256('InfinityExchange'),
        keccak256(bytes('1')), // for versionId = 1
        block.chainid,
        address(this)
      )
    );
    WETH = _WETH;
    MATCH_EXECUTOR = _matchExecutor;
  }
```

Recommendation

1. Add validation checks to ensure `_WETH` and `_matchExecutor` are not zero addresses:

```
require(_WETH != address(0), 'WETH cannot be zero');
require(_matchExecutor != address(0), 'Executor cannot be zero');
```

1. Mark `WETH` and `MATCH_EXECUTOR` as `immutable` to reduce gas costs and increase safety:

```
address public immutable WETH;
address public immutable MATCH_EXECUTOR;
```

14. Unrestricted Match Executor Update Without Validation or Delay

Severity	Low
Finding ID	BH-2022-06-infinity-14
Target	contracts/core/InfinityExchange.sol
Function name	updateMatchExecutor

Description

The `updateMatchExecutor` function allows the contract owner to set a new `MATCH_EXECUTOR` address, which is a privileged role responsible for executing matched orders. However, the function lacks any validation of the new address and does not implement any access delay or safety checks. This creates a centralization risk: a malicious or compromised owner could assign the role to a contract or account that executes invalid trades, steals funds, or manipulates order flow.

```
function updateMatchExecutor(address _matchExecutor) external onlyOwner {
    MATCH_EXECUTOR = _matchExecutor;
  }
```

Recommendation

1. Add validation to ensure `_matchExecutor` is not the zero address:

```
require(_matchExecutor != address(0), 'Invalid executor address');
```

1. (Optional but recommended) Introduce a time-lock or delay mechanism for critical role updates, or require multi-signature authorization to enhance security.

15. Lack of Upper Bound Checks on Penalty Update Allows Malicious Owner Behavior

Severity	Low
Finding ID	BH-2022-06-infinity-15
Target	contracts/staking/InfinityStaker.sol
Function name	updatePenalties

Description

The `updatePenalties` function allows the contract owner to update staking penalties for early withdrawal or other staking-related logic. However, it does not impose any upper or logical bounds on the `threeMonthPenalty`, `sixMonthPenalty`, and `twelveMonthPenalty` values. A malicious or compromised owner could set these penalties to extremely high values (e.g., 10000 or more in basis points), effectively preventing users from withdrawing their staked funds or unfairly slashing their assets. This introduces a major trust and centralization risk.

```
function updatePenalties(
    uint16 threeMonthPenalty,
    uint16 sixMonthPenalty,
    uint16 twelveMonthPenalty
  ) external onlyOwner {
    THREE_MONTH_PENALTY = threeMonthPenalty;
    SIX_MONTH_PENALTY = sixMonthPenalty;
    TWELVE_MONTH_PENALTY = twelveMonthPenalty;
  }
```

Recommendation

Add validation to ensure the penalties are within a reasonable range (e.g., 0 to 10000 basis points, equivalent to 0% to 100%):

```
require(threeMonthPenalty <= 10000, 'Penalty too high');
require(sixMonthPenalty <= 10000, 'Penalty too high');
require(twelveMonthPenalty <= 10000, 'Penalty too high');
```

Also consider documenting expected ranges and updating via a time-locked governance mechanism if decentralization is a goal.