SHERLOCK

# SHERLOCK SECURITY REVIEW FOR


**Prepared for:** bullvbear
**Prepared by:** Sherlock
**Lead Security Expert:** WATCHPUG
**Dates Audited:** November 14 - November 17, 2022
**Prepared on:** December 12, 2022

# Introduction

With Bull v Bear you can short NFT collections, hedge your portfolio and buy discounted NFTs. Soon, on Ethereum.

## Scope

`src/BvbProtocol.sol`

---

## Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

## Issues found

| Medium | High |
|---|---|
| 2 | 4 |

## Issues not fixed or acknowledged

| Medium | High |
|---|---|
| 0 | 0 |

## Security experts who found valid issues

ak1
carrot
bin2chen
WATCHPUG
kirk-baird
Bahurum
curiousapple
KingNFT
ElKu
GimelSec
hansfriese
neumo
0x52
Zarf
Ruhum
0x4non
__141345__
0xSmartContract
rvierdiiev
dipp
imare
aviggiano
cccz
0xmuxyz

obront
0xadrii

Tomo
tives

0v3rf10w
pashov

# Issue H-1: Attackers can use `reclaimContract()` to transfer assets in protocol to address(0)

Source: https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/127

## Found by

kirk-baird, bin2chen, GimelSec, 0x52, __141345__, Ruhum, carrot, hansfriese

## Summary

`reclaimContract()` would transfer payment tokens to `bulls[contractId]`. An attacker can make `reclaimContract()` transfer assets to address(0).

## Vulnerability Detail

An attacker can use a fake order to trick `reclaimContract()`. The fake order needs to meet the following requirements:

- `block.timestamp > order.expiry`.
- `!settledContracts[contractId]`.
- `!reclaimedContracts[contractId],`.

The first one is easy to fulfilled, an attacker can decide the content of the fake order. And the others are all satisfied since the fake order couldn't be settled or reclaimed before.

Thus, `reclaimContract()` would run this line: `IERC20(order.asset).safeTransfer(bull, bullAssetAmount);`. `bull` is address(0) since `bulls[contractId]` hasn't been filled. If `order.asset`'s implementation doesn't make sure `to != address(0)`(e.g., https://github.com/ConsenSys/Tokens/blob/fdf687c69d998266a95f15216b1955a4965a0a6d/contracts/eip20/EIP20.sol). The asset would be sent to address(0).

```
function reclaimContract(Order calldata order) public nonReentrant {
    bytes32 orderHash = hashOrder(order);

    // ContractId
    uint contractId = uint(orderHash);

    address bull = bulls[contractId];

    // Check that the contract is expired
    require(block.timestamp > order.expiry, "NOT_EXPIRED_CONTRACT");
```

```
    // Check that the contract is not settled
    require(!settledContracts[contractId], "SETTLED_CONTRACT");

    // Check that the contract is not reclaimed
    require(!reclaimedContracts[contractId], "RECLAIMED_CONTRACT");

    uint bullAssetAmount = order.premium + order.collateral;
    if (bullAssetAmount > 0) {
        // Transfer payment tokens to the Bull
        IERC20(order.asset).safeTransfer(bull, bullAssetAmount);
    }

    reclaimedContracts[contractId] = true;

    emit ReclaimedContract(orderHash, order);
}
```

## Impact

An attacker can use this vulnerability to transfer assets from BvB to address(0). It results in serious loss of funds.

## Code Snippet

https://github.com/sherlock-audit/2022-11-bullvbear/blob/main/bvb-protocol/src/BvbProtocol.sol#L417-L443

## Tool used

Manual Review

## Recommendation

There are multiple solutions for this problem.

1. check `bulls[contractId] != address(0)`
2. check the order is matched `matchedOrders[contractId].maker != address(0)`

## Discussion

**datschill**

PR fixing this issue : https://github.com/BullvBear/bvb-solidity/pull/4

**jack-the-pug**

Fix confirmed

# Issue H-2: Bull can `transferPosition()` to `address(0)` and the original order can be matched again

Source: https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/114

## Found by

bin2chen, imare, GimelSec, neumo, WATCHPUG, curiousapple, 0x52, rvierdiiev, aviggiano, Bahurum, KingNFT, dipp, carrot, hansfriese

## Summary

Using `bulls[uint(orderHash)] == address(0)` to check whether the order is matched is insufficient, the bull can `transferPosition` to `address(0)` and the order can be matched again.

## Vulnerability Detail

An order must not be matched more than once.

There is a check presented in the current implementation to prevent that: L760 `require(bulls[uint(orderHash)] == address(0), "ORDER_ALREADY_MATCHED");`.

However, this check can be easily bypassed by the bull, as they can `transferPosition()` to `address(0)` anytime.

Then the original order can be matched again.

## Impact

Attacker can match the orders by bear makers multiple times, pulling `order.premium + bearFees` from the victims' wallet as many times as they want.

## Code Snippet

https://github.com/sherlock-audit/2022-11-bullvbear/blob/main/bvb-protocol/src/BvbProtocol.sol#L734-L761

https://github.com/sherlock-audit/2022-11-bullvbear/blob/main/bvb-protocol/src/BvbProtocol.sol#L521-L538

## Tool used

Manual Review

## Recommendation

Consider using `matchedOrders[contractId]` to check if the order has been matched or not. Also, consider disallowing `transferPosition()` to `address(0)`.

## Discussion

**datschill**

PR fixing this issue : https://github.com/BullvBear/bvb-solidity/pull/1

**datschill**

PR fixing the transfer to 0x0 : https://github.com/BullvBear/bvb-solidity/pull/3

**jack-the-pug**

Fix confirmed

# Issue H-3: Bull can prevent `settleContract()`

Source: https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/111

## Found by

ak1, WATCHPUG, curiousapple, Bahurum, KingNFT, ElKu

## Summary

The bull can intentionally cause out-of-gas and revert the transaction and prevent `settleContract()`.

## Vulnerability Detail

As `IERC721(order.collection).safeTransferFrom()` is used in `settleContract()` which will call `IERC721Receiver(to).onERC721Received()` when the `to` address is an contract.

This gives the bull a chance to intentionally prevent the transaction from happening by consuming a lot of gas and revert the whole transaction.

## Impact

The bear (victim) can not `settleContract()` therefore cannot exercise their put option rights. The bull (attacker) always wins.

## Code Snippet

https://github.com/sherlock-audit/2022-11-bullvbear/blob/main/bvb-protocol/src/BvbProtocol.sol#L374-L411

## Tool used

Manual Review

## Recommendation

```
function settleContract(Order calldata order, uint tokenId) public nonReentrant {
    bytes32 orderHash = hashOrder(order);

    // ContractId
    uint contractId = uint(orderHash);

    address bear = bears[contractId];
```

```
    // Check that only the bear can settle the contract
    require(msg.sender == bear, "ONLY_BEAR");

    // Check that the contract is not expired
    require(block.timestamp < order.expiry, "EXPIRED_CONTRACT");

    // Check that the contract is not already settled
    require(!settledContracts[contractId], "SETTLED_CONTRACT");

    address bull = bulls[contractId];

-    // Try to transfer the NFT to the bull (needed in case of a malicious bull
 ↪  that block transfers)
-    try IERC721(order.collection).safeTransferFrom(bear, bull, tokenId) {}
-    catch (bytes memory) {
        // Transfer NFT to BvbProtocol
        IERC721(order.collection).safeTransferFrom(bear, address(this), tokenId);
        // Store that the bull has to retrieve it
        withdrawableCollectionTokenId[order.collection][tokenId] = bull;
-    }

    uint bearAssetAmount = order.premium + order.collateral;
    if (bearAssetAmount > 0) {
        // Transfer payment tokens to the Bear
        IERC20(order.asset).safeTransfer(bear, bearAssetAmount);
    }

    settledContracts[contractId] = true;

    emit SettledContract(orderHash, tokenId, order);
}
```

## Discussion

**datschill**

PR fixing this issue : https://github.com/BullvBear/bvb-solidity/pull/14

**sherlock-admin**

> Escalate for 5 USDC
>
> Hey Hi, From all issues considered duplicates here, this current issue https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/111 and https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/100 are incorrect, and shouldn't be considered. Both of https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/111,

https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/100 say that bulls can stop bears from settling via out-of-gas revert.

However, Bulls can not directly stop bears from settling, as these 2 reports depict. All they can do is increase transaction costs and add extra overhead, as correctly explained in

https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/147
https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/18
https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/142

Verification Copy follwoing test inside `bvb-protocol> test> integration>` and run `forge test --match-contract TestGasGrief -vvvvv` https://gist.github.com/abhishekvispute/26bc7e0e231d1ca3cf4deae2dd5d2ebf

In this test, you will see that the bull transfers the position to a malicious bull which has an infinite loop on received, but still bear can settle by paying additional overhead. Hence the issue is not that bulls can cause out-of-gas reverts but bulls adding additional gas overhead. The recommendation is also wrong for https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/111. **Consider removing them from included issues.**

test trace (check how it comes out of "out of gas revert" and still allows to settle)

```
[PASS] testGasGrief((uint256,uint256,uint256,uint256,uint256,uint16,address
↪  ,address,address,bool)) (runs: 256, : 7301975, ~: 7304974)
Traces:
 .......
 .......
 .......
    [0] VM::prank(Bear: [0x7B5969C684b101DA876186F1b8f3aB808e308B7c])
      ← ()
    [6672810] BvbProtocol::settleContract((4891, 864, 3601, 86401, 10, 20,
    ↪  0x7B5969C684b101DA876186F1b8f3aB808e308B7c,
    ↪  0xCe71065D4017F316EC606Fe4422e11eB2c47c246,
    ↪  0x185a4dc360CE69bDCceE33b3784B0282f7961aea, false), 1234)
       [6533561] BvbERC721::safeTransferFrom(Bear:
    ↪  [0x7B5969C684b101DA876186F1b8f3aB808e308B7c], BvbMaliciousBull:
    ↪  [0x42997aC9251E5BB0A61F4Ff790E5B991ea07Fd9B], 1234)
          emit Transfer(from: Bear:
    ↪  [0x7B5969C684b101DA876186F1b8f3aB808e308B7c], to: BvbMaliciousBull:
    ↪  [0x42997aC9251E5BB0A61F4Ff790E5B991ea07Fd9B], id: 1234)
          [6522355] BvbMaliciousBull::onERC721Received(BvbProtocol:
    ↪  [0xf5a2fE45F4f1308502b1C136b9EF8af136141382], Bear:
    ↪  [0x7B5969C684b101DA876186F1b8f3aB808e308B7c], 1234, 0x)
             ← "EvmError: OutOfGas"
          ← "EvmError: Revert"
```

```
    [23775] BvbERC721::safeTransferFrom(Bear:
↪   [0x7B5969C684b101DA876186F1b8f3aB808e308B7c], BvbProtocol:
↪   [0xf5a2fE45F4f1308502b1C136b9EF8af136141382], 1234)
       emit Transfer(from: Bear:
↪   [0x7B5969C684b101DA876186F1b8f3aB808e308B7c], to: BvbProtocol:
↪   [0xf5a2fE45F4f1308502b1C136b9EF8af136141382], id: 1234)
       [864] BvbProtocol::onERC721Received(BvbProtocol:
↪   [0xf5a2fE45F4f1308502b1C136b9EF8af136141382], Bear:
↪   [0x7B5969C684b101DA876186F1b8f3aB808e308B7c], 1234, 0x)
          ← 0x150b7a02
       ← ()
    [22852] BvbERC20::transfer(Bear:
↪   [0x7B5969C684b101DA876186F1b8f3aB808e308B7c], 5755)
       emit Transfer(from: BvbProtocol:
↪   [0xf5a2fE45F4f1308502b1C136b9EF8af136141382], to: Bear:
↪   [0x7B5969C684b101DA876186F1b8f3aB808e308B7c], amount: 5755)
       ← true
    emit SettledContract(orderHash:
↪   0xf737abeb07bf156db35b7a738422241312110236659091d877b6d09846af2e82,
↪   tokenId: 1234, order: (4891, 864, 3601, 86401, 10, 20,
↪   0x7B5969C684b101DA876186F1b8f3aB808e308B7c,
↪   0xCe71065D4017F316EC606Fe4422e11eB2c47c246,
↪   0x185a4dc360CE69bDCceE33b3784B0282f7961aea, false))
    emit log_named_uint(key: safeTransferCase Gas, val: 107483)
    ← ()
 [564] BvbERC20::balanceOf(Bull:
↪   [0xCf03Dd0a894Ef79CB5b601A43C4b25E3Ae4c67eD]) [staticcall]
    ← 0
 [564] BvbERC20::balanceOf(Bear:
↪   [0x7B5969C684b101DA876186F1b8f3aB808e308B7c]) [staticcall]
    ← 5755
 ← ()
```

You've deleted an escalation for this issue.

**jack-the-pug**

Fix confirmed

# Issue H-4: Reentrancy in `withdrawToken()` May Delete The Next User's Balance

Source: https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/88

## Found by

kirk-baird, ak1, bin2chen, 0x4non, neumo, 0xSmartContract, carrot, Zarf

## Summary

The function `withdrawToken()` does not have a reentrancy guard and calls an external contract. It is possible to reenter `settleContract()` to spend the same token that was just transferred out. If the `safeTransferFrom()` in `settleContract()` fails then the token balance is added to the bull. However, when `withdrawToken()` continues execution it will delete the balance of the bull.

## Vulnerability Detail

`withdrawToken()` makes a state change to `withdrawableCollectionTokenId[collection][tokenId]` after it makes an external call to an ERC721 contract `safeTransferFrom()`. Since this external call will relinquish control to the `to` address which is `recipient`, the `recipient` smart contract may reenter `settleContract()`.

When calling `settleContract()` set the `tokenId` function parameter to the same one just transferred in `withdawToken()`. If transfer to the `bull` fails then the token is instead transferred to `BvbProtocol` and balance added to the bull, `withdrawableCollectionTokenId[order.collection][tokenId] = bull`

After `settleContract()` finishes executing control will revert back to `withdrawToken()` which then executes the line `withdrawableCollectionTokenId[collection][tokenId] = address(0)`. The balance of the bull is therefore delete for that token.

e.g. If we know a transfer will fail to a bull in a matched order we can a) create a fake order with ourselves b) reenter from `withdrawToken()` into `settleContract()` and therefore delete the bulls `withdrawableCollectionTokenId` balance. Steps:

- `BvpProtocol.matchOrder(orderA)` create a fake order (A) with ones self
- `BvpProtocol.settleOrder(orderA)` settle the fake order (A) with ones self and ensure the ERC721 transfer from bull to bear fails.
- `BvpProtocol.matchOrder(orderB)` match the real order (B), this can be done at any time

- `BvbProtocol.withdrawToken(orderA, token1)` the following setups happen during line #456
  - `ERC721(collection).safeTransferFrom(this, recipient, tokenId)` (`recipient` is bull from the fake order (A))
  - `recipient.onERC721Received()` called by `safeTransferFrom()` and gives execution control to `receipient`
  - `BvpProtocol.settleOrder(orderB, token1)` reenter to settle the real order using `token1` which does `withdrawableCollectionTokenId[order.collection][tokenId] = bull`
- Finish executing `BvbProtocol.withdrawToken(orderA, token1)` after line #456 which does `withdrawableCollectionTokenId[collection][tokenId] = address(0)`

## Impact

If we know a transfer is going to fail to a `bull` for an ERC721 we can ensure the NFT is locked in the `BvbProtocol` contract. This NFT will be unrecoverable.

## Code Snippet

withdrawToken()

```
function withdrawToken(bytes32 orderHash, uint tokenId) public {
    address collection = matchedOrders[uint(orderHash)].collection;

    address recipient = withdrawableCollectionTokenId[collection][tokenId];

    // Transfer NFT to recipient
    IERC721(collection).safeTransferFrom(address(this), recipient, tokenId);

    // This token is not withdrawable anymore
    withdrawableCollectionTokenId[collection][tokenId] = address(0);
```

settleContract()

```
function settleContract(Order calldata order, uint tokenId) public nonReentrant {
    bytes32 orderHash = hashOrder(order);

    // ContractId
    uint contractId = uint(orderHash);

    address bear = bears[contractId];

    // Check that only the bear can settle the contract
```

```
    require(msg.sender == bear, "ONLY_BEAR");

    // Check that the contract is not expired
    require(block.timestamp < order.expiry, "EXPIRED_CONTRACT");

    // Check that the contract is not already settled
    require(!settledContracts[contractId], "SETTLED_CONTRACT");

    address bull = bulls[contractId];

    // Try to transfer the NFT to the bull (needed in case of a malicious bull
↪   that block transfers)
    try IERC721(order.collection).safeTransferFrom(bear, bull, tokenId) {}
    catch (bytes memory) {
        // Transfer NFT to BvbProtocol
        IERC721(order.collection).safeTransferFrom(bear, address(this), tokenId);
        // Store that the bull has to retrieve it
        withdrawableCollectionTokenId[order.collection][tokenId] = bull;
    }

    uint bearAssetAmount = order.premium + order.collateral;
    if (bearAssetAmount > 0) {
        // Transfer payment tokens to the Bear
        IERC20(order.asset).safeTransfer(bear, bearAssetAmount);
    }

    settledContracts[contractId] = true;

    emit SettledContract(orderHash, tokenId, order);
}
```

## Tool used

Manual Review

## Recommendation

I recommend both of these solutions though either one will be sufficient on its own:

- Add `nonReentrant` modifier to `withdrawToken()`
- Set `withdrawableCollectionTokenId[collection][tokenId] = address(0)` before performing `IERC721(collection).safeTransferFrom(address(this), recipient, tokenId)` to apply the checks-effects-interactions pattern.

## Discussion

**datschill**

PR fixing checks-effects-interactions pattern :
https://github.com/BullvBear/bvb-solidity/pull/15

**datschill**

PR fixing another issue, removing the withdrawToken() method :
https://github.com/BullvBear/bvb-solidity/pull/14

**sherlock-admin**

> Escalate for 1 USDC
>
> Reason There is no working re-entrancy attack path, the risk level should be LOW.
>
> (1) The recipient has been changed to victim bull while re-entry 'settleContract' described in submission #77, #88, a next call from attacker to 'withdrawToken' will fail
>
> (2) The underlying token has been transfered to attacker bull, so re-entry 'withdrawToken' decribed in submission #8 would not work too
>
> (3) The last re-entry point is 'transferPosition', no damage too
>
> Test case

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.17;

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {Base} from "./Base.t.sol";

import {BvbProtocol} from "src/BvbProtocol.sol";
import "forge-std/console.sol";

contract Victim {
    // Some collection contracts limit max number of NFTs an account can
↪   hold.
    // This contract simulates an account subjected to the limit, that is
↪   the account can't receive
    // NFT now, but later if it sends/sells some NFTs out, it can receive
↪   NFT again, withdraw NTFs
    // which are kept in BvbProtocol due to previous limit.
    bool private _limited;

    function setLimited(bool limited) external {
        _limited = limited;
    }
```

```
    function onERC721Received(
        address ,
        address ,
        uint256 ,
        bytes calldata
    ) external returns (bytes4) {
        require(!_limited);
        return this.onERC721Received.selector;
    }

    function withdrawToken(address bvb, bytes32 orderHash, uint tokenId)
↪   external {
        BvbProtocol(bvb).withdrawToken(orderHash, tokenId);
    }
}


contract BadBearsAttackContract {
    bool private attack;
    bool private receiveNFT;
    address private owner;
    address private target;
    uint private tokenId;
    bytes32 private contractId;
    BvbProtocol.Order private order;

    constructor () {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function enableAttack(address _target, bytes32 _contractId, uint
↪   _tokenId, BvbProtocol.Order calldata _order) external onlyOwner {
        attack = true;
        target = _target;
        contractId = _contractId;
        tokenId = _tokenId;
        order = _order;
    }

    function enableReceive(bool _receive) external onlyOwner {
        receiveNFT = _receive;
    }
```

```
    function onERC721Received(
        address ,
        address ,
        uint256 id,
        bytes calldata
    ) external returns (bytes4) {
        require(receiveNFT);
        if (attack && tokenId == id) {
            attack = false;
            BvbProtocol(target).settleContract(order, id);
            BvbProtocol(target).withdrawToken(contractId, id);
        }

        return this.onERC721Received.selector;
    }
}

contract ExploitWithdrawReentrancy is Base {
    Victim internal victim;
    BadBearsAttackContract internal attack;

    function setUp() public {
        victim = new Victim();
        attack = new BadBearsAttackContract();

        bvb.setAllowedAsset(address(weth), true);
        bvb.setAllowedCollection(address(doodles), true);

        deal(address(weth), bull, 0xffffffff);
        deal(address(weth), bear, 0xffffffff);
        deal(address(weth), address(victim), 0xffffffff);
        deal(address(weth), address(attack), 0xffffffff);
        vm.prank(bull);
        weth.approve(address(bvb), type(uint).max);
        vm.prank(bear);
        weth.approve(address(bvb), type(uint).max);
        vm.prank(address(victim));
        weth.approve(address(bvb), type(uint).max);
        vm.prank(address(attack));
        weth.approve(address(bvb), type(uint).max);
    }

    function testExploitWithdrawReentrancy() public {
        BvbProtocol.Order memory order = defaultOrder();
        order.maker = bear;
        order.isBull = false;

        bytes32 orderHash = bvb.hashOrder(order);
```

```
        // Sign the order
        bytes memory signature = signOrderHash(bearPrivateKey, orderHash);

        // Taker (Bull) match with this order
        vm.prank(address(victim));
        bvb.matchOrder(order, signature);

        // Give a NFT to the Bear + approve
        uint tokenId = 1234;
        doodles.mint(bear, tokenId);
        vm.prank(bear);
        doodles.setApprovalForAll(address(bvb), true);

        // Bad bear create a new order with the same collection but earlier
↪   expiry, and match with self's attack contract
        BvbProtocol.Order memory order2 = defaultOrder();
        order2.maker = bear;
        order2.isBull = false;
        order2.expiry = order.expiry - 1 days;
        bytes32 orderHash2 = bvb.hashOrder(order2);
        bytes memory signature2 = signOrderHash(bearPrivateKey, orderHash2);
        vm.prank(address(attack));
        bvb.matchOrder(order2, signature2);

        vm.prank(bear);
        bvb.settleContract(order2, tokenId);
        assertEq(bvb.withdrawableCollectionTokenId(address(doodles),
↪   tokenId), address(attack), "Token kept for badBear");
        vm.prank(address(attack));
        doodles.setApprovalForAll(address(bvb), true);


        // transfer the previous position to attack contract
        vm.prank(bear);
        bvb.transferPosition(orderHash, false, address(attack));

        attack.enableReceive(true);
        attack.enableAttack(address(bvb), orderHash2, tokenId, order);

        victim.setLimited(true);

        vm.expectRevert();
        vm.prank(address(attack));
        bvb.withdrawToken(orderHash2, tokenId);
    }
}
```

You've deleted an escalation for this issue.

**jack-the-pug**

Fix confirmed

# Issue M-1: It doesn't handle fee-on-transfer/deflationary tokens

Source: https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/130

## Found by

Tomo, GimelSec, pashov, rvierdiiev, 0v3rf10w, cccz, dipp, Ruhum, tives, Zarf, hansfriese

## Summary

The protocol doesn't handle fee-on-transfer/deflationary tokens, users will be unable to call `settleContract` and `reclaimContract` due to not enough assets in the contract. Though the protocol uses `allowedAsset` to set the asset as supported as payment, we can't guarantee that the allowed non-deflationary token will always not become a deflationary token, especially upgradeable tokens (for example, USDC).

## Vulnerability Detail

Assume that A token is a deflationary token, and it will take 50% fee when transferring tokens. And the protocol only set 4% fee.

If a user is bear and call `mathOrder` with `order.premium = 100`, the `takerPrice` will be `100 + 100*4% = 104` but the protocol will only get `104 * 50% = 52` tokens in L354. Same problem in `order.collateral`, the user will be unable to call `settleContract` because the contract doesn't have enough A tokens.

## Impact

The protocol will be unable to pay enough tokens to users when users want to call `settleContract` or `reclaimContract`.

## Code Snippet

https://github.com/sherlock-audit/2022-11-bullvbear/blob/main/bvb-protocol/src/BvbProtocol.sol#L354 https://github.com/sherlock-audit/2022-11-bullvbear/blob/main/bvb-protocol/src/BvbProtocol.sol#L358

## Tool used

Manual Review

## Recommendation

Use `balanceAfter - balanceBefore`:

```
uint256 balanceBefore = deflationaryToken.balanceOf(address(this));
deflationaryToken.safeTransferFrom(msg.sender, address(this), takerPrice);
uint256 balanceAfter = deflationaryToken.balanceOf(address(this));
premium = (balanceAfter - balanceBefore) - bearFees;
```

## Discussion

**datschill**

PR fixing this issue : https://github.com/BullvBear/bvb-solidity/pull/8

**jack-the-pug**

Fix confirmed

# Issue M-2: Bulls that are unable to receive NFTs will not be able to claim them later

Source: https://github.com/sherlock-audit/2022-11-bullvbear-judging/issues/4

## Found by

bin2chen, GimelSec, rvierdiiev, WATCHPUG, 0xmuxyz, 0xadrii, cccz, carrot, obront, hansfriese

## Summary

A lot of care has been taken to ensure that, if a bull has a contract address that doesn't accept ERC721s, the NFT is saved to `withdrawableCollectionTokenId` for later withdrawal. However, because there is no way to withdraw this token to a different address (and the original address doesn't accept NFTs), it will never be able to be claimed.

## Vulnerability Detail

To settle a contract, the bear calls `settleContract()`, which sends their NFT to the bull, and withdraws the collateral and premium to the bear.

```
try IERC721(order.collection).safeTransferFrom(bear, bull, tokenId) {}
catch (bytes memory) {
    // Transfer NFT to BvbProtocol
    IERC721(order.collection).safeTransferFrom(bear, address(this), tokenId);
    // Store that the bull has to retrieve it
    withdrawableCollectionTokenId[order.collection][tokenId] = bull;
}

uint bearAssetAmount = order.premium + order.collateral;
if (bearAssetAmount > 0) {
    // Transfer payment tokens to the Bear
    IERC20(order.asset).safeTransfer(bear, bearAssetAmount);
}
```

In order to address the case that the bull is a contract that can't accept NFTs, the protocol uses a try-catch setup. If the transfer doesn't succeed, it transfers the NFT into the contract, and sets `withdrawableCollectionTokenId` so that the specific NFT is attributed to the bull for later withdrawal.

However, assuming the bull isn't an upgradeable contract, this withdrawal will never be possible, because their only option is to call the same function `safeTransferFrom` to the same contract address, which will fail in the same way.

```
function withdrawToken(bytes32 orderHash, uint tokenId) public {
    address collection = matchedOrders[uint(orderHash)].collection;

    address recipient = withdrawableCollectionTokenId[collection][tokenId];

    // Transfer NFT to recipient
    IERC721(collection).safeTransferFrom(address(this), recipient, tokenId);

    // This token is not withdrawable anymore
    withdrawableCollectionTokenId[collection][tokenId] = address(0);

    emit WithdrawnToken(orderHash, tokenId, recipient);
}
```

## Impact

If a bull is a contract that can't receive NFTs, their orders will be matched, the bear will be able to withdraw their assets, but the bull's NFT will remain stuck in the BVB protocol contract.

## Code Snippet

https://github.com/sherlock-audit/2022-11-bullvbear/blob/main/bvb-protocol/src/BvbProtocol.sol#L394-L406

https://github.com/sherlock-audit/2022-11-bullvbear/blob/main/bvb-protocol/src/BvbProtocol.sol#L450-L462

## Tool used

Manual Review

## Recommendation

There are a few possible solutions:

- Add a `to` field in the `withdrawToken` function, which allows the bull to withdraw the NFT to another address
- Create a function similar to `transferPosition` that can be used to transfer owners of a withdrawable NFT
- Decide that you want to punish bulls who aren't able to receive NFTs, in which case there is no need to save their address or implement a `withdrawToken` function

## Discussion

**datschill**

PR fixing another issue, removing the withdrawToken() method : https://github.com/BullvBear/bvb-solidity/pull/14

**datschill**

This issue isn't High, because in the default behavior, no smart contract can match an Order. So for a Bull to be a smart contract, the user needs to match an order (as a maker or a taker) with an EOA, then transfer his position to a smart contract. This would be kind of a poweruser move, so we consider that he should be aware that his smart contract should handle NFT reception. Whatsoever, the issue is fixed thanks to the PR#14, the user will be able to transfer his position to whatever EOA or smart contract he wants before calling reclaimContract() to retrieve ERC20 assets or ERC721.

**jack-the-pug**

Fix confirmed. The bull who wish to withdraw the NFT to another address shall call `transferPosition()` before `reclaimContract()`.