BID-BEASTS Security Review

Reviewer: Muhammad Faran **Date:** January 19, 2025 **Repo:** CodeHawks-Contests/2025-09-bid-beasts

---

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold 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 view of the code was solely on the security aspects of the Solidity implementation of the contracts.

About the Project

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.

Project Scope

```
├── lib/
├── src/
│   ├── BidBeasts_NFT_ERC721.sol
│   └── BidBeastsNFTMarketPlace.sol
```

Actors

- **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.

FINDING 1: Root + Impact

Description

The protocol is intended to implement standard ERC721 functionality, where critical state-changing actions like burning (destroying) a token are restricted to the token's owner or an approved operator.

However, the public `burn` function acts as a wrapper for the internal `_burn` function without implementing any access control checks. As noted in the OpenZeppelin ERC721 documentation, the internal `_burn` function "does not check if the sender is authorized to operate on the token." Consequently, any external actor can call this function on any existing `tokenId` to destroy it.

```
function burn(uint256 _tokenId) public {
    // @> Root cause: Has no access control; no require check precedes it.
    _burn(_tokenId);

    emit BidBeastsBurn(msg.sender, _tokenId);
}
```

Risk

Likelihood: High

- The function is `public` and exposed to all external users without restriction.
- The attack requires no special setup or capital; an attacker simply needs to know a valid `tokenId`.

Impact: High

- Malicious actors can permanently destroy (burn) NFTs owned by other users.
- This constitutes a critical griefing vector resulting in the permanent loss of user assets.

Proof of Concept

The following Foundry test demonstrates that an arbitrary attacker can successfully burn a token they do not own.

```
function test_anyoneCanBurnBidBeastNFT() public {
    uint256 tokenId = 0;

    // 1. Setup: Mint an NFT to the default user/owner
```

```
    _mintNFT();

    // 2. Action: An arbitrary attacker calls the public burn function
    vm.prank(ATTACKER);
    nft.burn(tokenId);

    // 3. Assertion: The token no longer exists (ownerOf reverts)
    vm.expectRevert();
    nft.ownerOf(tokenId);
}
```

Recommended Mitigation

Restrict the `burn` function so that only the token owner or an approved operator can execute it.

```
    function burn(uint256 _tokenId) public {
+       require(_isApprovedOrOwner(_msgSender(), _tokenId), "ERC721: caller is not owner nor approved");
        _burn(_tokenId);

        emit BidBeastsBurn(msg.sender, _tokenId);
    }
```

FINDING 2: Root + Impact

Description

- The protocol documentation and intended business logic state that an auction should last for 3 days after the first bid is placed.
- However, when the first bid is placed, the code initializes the `auctionEnd` timestamp using `S_AUCTION_EXTENSION_DURATION` (15 minutes) instead of the intended full auction duration.

```
    if (previousBidAmount == 0) {
        requiredAmount = listing.minPrice;
        require(msg.value > requiredAmount, "First bid must be > min price");
        // @> Root cause: Auction is set to end in 15 minutes (extension duration) instead of 3 days
        listing.auctionEnd = block.timestamp + S_AUCTION_EXTENSION_DURATION;
        emit AuctionExtended(tokenId, listing.auctionEnd);
    }
```

Risk

Likelihood: High

- This will occur on **every single auction** initiated in the protocol, as the logic is inside the primary `placeBid` function and executes unconditionally for the first bid.

Impact: Low

- **Significant loss of value for sellers**: Auctions will close prematurely (after 15 minutes), preventing legitimate price discovery and potential bidding wars that would occur over a 3-day period.
- **Protocol malfunction**: The marketplace fails to meet its core functional specification of providing 3-day auctions.

Proof of Concept

The following test confirms that after the first bid, the auction end time is set to `block.timestamp + 15 minutes` rather than the expected `3 days`.

```
function test_auctionDurationNotThreeDays() public {
    uint256 BUY_PRICE = 2 ether;
    _mintNFT();
    _listNFT();

    vm.prank(BIDDER_1);
    market.placeBid{value: BUY_PRICE}(TOKEN_ID);

    BidBeastsNFTMarket.Bid memory highestBid =
market.getHighestBid(TOKEN_ID);
    assertEq(highestBid.bidder, BIDDER_1);
    assertEq(highestBid.amount, BUY_PRICE);

    // Check that the auction end matches the extension duration (15 mins)
    assertEq(market.getListing(TOKEN_ID).auctionEnd, block.timestamp +
market.S_AUCTION_EXTENSION_DURATION());

    console.log("Time", market.getListing(TOKEN_ID).auctionEnd);

    // Proves the auction is significantly shorter than 3 days
    assertTrue(market.getListing(TOKEN_ID).auctionEnd < block.timestamp + 3
days);
}
```

Recommended Mitigation

Define a new constant for the initial auction duration (e.g., 3 days) and use it when initializing the auction on the first bid.

```
+   uint256 public constant S_AUCTION_DURATION = 3 days;

    // ... inside placeBid ...
```

```
    if (previousBidAmount == 0) {
        requiredAmount = listing.minPrice;
        require(msg.value > requiredAmount, "First bid must be > min price");

-       listing.auctionEnd = block.timestamp +
S_AUCTION_EXTENSION_DURATION;
+       listing.auctionEnd = block.timestamp + S_AUCTION_DURATION;
        emit AuctionExtended(tokenId, listing.auctionEnd);
    }
```

FINDING 3: Root + Impact

Description

The withdrawAllFailedCredits function is intended to allow users to retrieve funds that were previously credited to them after a failed transfer.

However, the function contains two critical flaws:

1. **Missing Access Control:** It allows any arbitrary caller (msg.sender) to specify any _receiver address to withdraw funds for.
2. **Incorrect State Update:** It reads the balance from the _receiver but resets the balance of the msg.sender.

This means an attacker can call this function passing a victim's address as _receiver. The contract will read the victim's positive balance but zero out the attacker's balance (which is already zero). The funds are then sent to the attacker (msg.sender), leaving the victim's balance in failedTransferCredits untouched, but draining the contract balance.

```
    function withdrawAllFailedCredits(address _receiver) external {
        // @> Reads amount from the victim (_receiver)
        uint256 amount = failedTransferCredits[_receiver];
        require(amount > 0, "No credits to withdraw");

        // @> Resets the ATTACKER'S balance, leaving the victim's balance FULL
        failedTransferCredits[msg.sender] = 0;

        // @> Sends the victim's funds to the ATTACKER
        (bool success, ) = payable(msg.sender).call{value: amount}("");
        require(success, "Withdraw failed");
    }
```

Risk

Likelihood: High

- The function is external and requires no special privileges.
- The attack is trivial to execute; an attacker simply needs to identify any user with a positive failedTransferCredits balance (which is public information).

Impact: Critical

- **Theft of Funds:** Attackers can steal all pending credits belonging to other users.
- **Double Spending / Protocol Draining:** Because the victim's failedTransferCredits are never set to zero, the victim (or the attacker again) can withdraw the same funds repeatedly, draining the entire contract's ETH balance.

Proof of Concept

The following exploit demonstrates how an attacker can steal Alice's funds while leaving her credit balance intact (allowing for repeated theft).

```
// Assume Alice has 10 ETH in failedTransferCredits

// Attacker contract
contract Exploit {
    BidBeastsNFTMarket target;

    constructor(address _target) {
        target = BidBeastsNFTMarket(_target);
    }

    function attack(address victim) external {
        // Steal victim's failed transfer credits
        target.withdrawAllFailedCredits(victim);
    }

    receive() external payable {}
}

// Attack flow:
// 1. Deploy Exploit contract
// 2. Call attack(alice_address)
// 3. The contract reads Alice's 10 ETH.
// 4. The contract sets Exploit's credits to 0 (Alice's credits remain 10 ETH).
// 5. The contract sends 10 ETH to Exploit.
// 6. Alice still has 10 ETH recorded in the mapping and can be exploited again.
```

Recommended Mitigation

The function should not accept an argument for the receiver. It should only allow the caller to withdraw their own funds, and it must update the correct balance *before* the transfer.

```
-   function withdrawAllFailedCredits(address _receiver) external {
+   function withdrawAllFailedCredits() external {
-       uint256 amount = failedTransferCredits[_receiver];
+       uint256 amount = failedTransferCredits[msg.sender];
        require(amount > 0, "No credits to withdraw");

-       failedTransferCredits[msg.sender] = 0;
+       failedTransferCredits[msg.sender] = 0;

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

FINDING 4: Root + Impact

Description

Under normal behavior, the AuctionSettled event should only be emitted **after an auction has actually concluded**, meaning the winning bid has been finalized, the NFT has been transferred, and seller proceeds have been paid.

In the current implementation of placeBid, the AuctionSettled event is emitted **before any auction settlement occurs** and even during regular bid placement. At this point, ownership has not changed, funds have not been distributed, and the auction remains active. This causes the event to misrepresent the actual protocol state.

```
function placeBid(uint256 tokenId) external payable isListed(tokenId) {
    ...
    require(msg.sender != previousBidder, "Already highest bidder");

    // @> Root cause: AuctionSettled is emitted during bid placement,
    // @> even though the auction has not ended or been finalized.
    emit AuctionSettled(tokenId, msg.sender, listing.seller, msg.value);

    // --- Regular Bidding Logic ---
    ...
}
```

Risk

Likelihood: HIGH

- Occurs on every successful call to placeBid that does not follow the buy-now path
- Independent of auction state, timing, or bid validity beyond basic checks

Impact: LOW

- Off-chain indexers, analytics systems, and frontends may incorrectly assume the auction has been finalized
- Monitoring, accounting, or automation systems relying on AuctionSettled may trigger premature or incorrect actions (e.g., marking NFTs as sold)

Proof of Concept

// Not required — event emission is unconditional and observable

Recommended Mitigation

Ensure that AuctionSettled is only emitted **after a successful auction settlement**, such as within _executeSale or settleAuction, and remove the premature emission from placeBid.

```
function placeBid(uint256 tokenId) external payable isListed(tokenId) {
    ...
    require(msg.sender != previousBidder, "Already highest bidder");

-   emit AuctionSettled(tokenId, msg.sender, listing.seller, msg.value);

    // --- Regular Bidding Logic ---
    ...
}
```

Optionally, introduce a separate event such as BidAccepted or HighestBidUpdated to represent successful bid placement without conflating it with auction finalization.

FINDING 5: Root + Impact

Description

- The auction’s minimum increment logic suffers from **integer division precision loss** because the contract divides before multiplying:

- This means bidders can submit **underpriced bids** that the contract accepts, undermining the auction's increment protection and systematically reducing seller revenue.

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

Risk

Likelihood: Medium

- The bug triggers whenever bids are not divisible.
- Attackers can deliberately exploit this to save ETH while still winning.

Impact: Low

- Sellers consistently lose value with each accepted underpriced bid.
- Auction integrity is broken, as increment rules are not enforced correctly.

Proof of Concept

This test demonstrates a precision loss vulnerability in the bidding system's minimum increment calculation. The test proves the vulnerability exists by comparing the correct calculation with the contract's flawed implementation.

```
function test_precisionLoss() public {
    _mintNFT();
    _listNFT();

    // Fund bidders
    vm.deal(BIDDER_1, 1 ether);
    vm.deal(BIDDER_2, 1 ether);

    // BIDDER_1 places first bid of 0.011 ETH (> 0.01 ETH min price)
    vm.prank(BIDDER_1);
    market.placeBid{value: 0.011 ether}(TOKEN_ID);

    // Pull increment percentage
    uint256 incPct = market.S_MIN_BID_INCREMENT_PERCENTAGE();

    // Current highest bid
    BidBeastsNFTMarket.Bid memory current = market.getHighestBid(TOKEN_ID);

    // Correct next min (multiply-first)
    uint256 expectedNext = (current.amount * (100 + incPct)) / 100;
```

```
    // Buggy contract calc (divide-first)
    uint256 contractCalc = (current.amount / 100) * (100 + incPct);

    // Logs for clarity
    emit log_named_uint("Current bid", current.amount);
    emit log_named_uint("Expected next min", expectedNext);
    emit log_named_uint("Contract-calculated next min", contractCalc);

    // Make sure contractCalc < expectedNext
    assertTrue(contractCalc < expectedNext, "Bug not triggered");

    // BIDDER_2 exploits by bidding contractCalc (cheaper than expectedNext)
    vm.prank(BIDDER_2);
    market.placeBid{value: contractCalc}(TOKEN_ID);

    // Verify BIDDER_2 is now highest bidder despite underpaying
    BidBeastsNFTMarket.Bid memory highest = market.getHighestBid(TOKEN_ID);
    assertEq(highest.bidder, BIDDER_2);
    assertEq(highest.amount, contractCalc);
    assertTrue(highest.amount < expectedNext, "Accepted bid was less than true
minimum increment");
}
```

Recommended Mitigation

Replace the vulnerable calculation with the mathematically correct version that performs multiplication before division:

```
- requiredAmount = (previousBidAmount / 100) * (100 +
S_MIN_BID_INCREMENT_PERCENTAGE);
+ requiredAmount = (previousBidAmount * (100 +
S_MIN_BID_INCREMENT_PERCENTAGE)) / 100;
```

FINDING 6:Root + Impact

Description

Under normal behavior, protocol fees and seller proceeds should be calculated in a way that is **precise, predictable, and consistent** with the documented fee percentage across all possible bid values.

In the current implementation of `_executeSale`, the protocol fee is calculated using integer division. Because Solidity performs **floor division**, this calculation may truncate fractional values, leading to minor rounding

discrepancies. As a result, the protocol may collect slightly less than the intended fee, and seller proceeds may be marginally higher than expected for certain bid amounts.

```
function _executeSale(uint256 tokenId) internal {
    ...
    // @> Root cause: integer division truncates fractional fees
    uint256 fee = (bid.amount * S_FEE_PERCENTAGE) / 100;
    s_totalFee += fee;

    uint256 sellerProceeds = bid.amount - fee;
    _payout(listing.seller, sellerProceeds);
    ...
}
```

Risk

Likelihood:

- Occurs whenever bid.amount * S_FEE_PERCENTAGE is not perfectly divisible by 100
- Common for bid amounts that are not multiples of 100 / S_FEE_PERCENTAGE

Impact:

- Protocol fee collection may be marginally lower than the nominal percentage
- Minor accounting discrepancies between expected and actual fee totals over time

---

Proof of Concept

// Not required — behavior is deterministic due to Solidity integer division

---

Recommended Mitigation

If higher precision or strict fee accounting is desired, consider using a higher-precision denominator or basis-point style math to reduce truncation effects.

```
- uint256 fee = (bid.amount * S_FEE_PERCENTAGE) / 100;
+ uint256 fee = (bid.amount * S_FEE_PERCENTAGE) / 10_000; // use basis points
```

Alternatively, explicitly document that fees are **rounded down** to avoid ambiguity between implementation and specification.

Auditor’s Note

This issue does **not** pose a security risk or fund-loss vector and is therefore correctly categorized as **Informational**. It primarily affects precision and accounting expectations rather than protocol safety.
