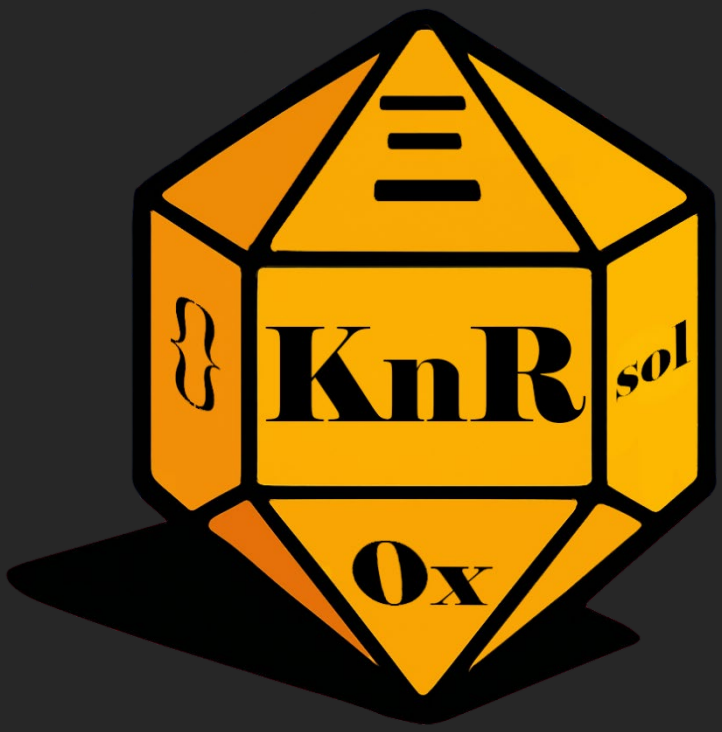

Kode-n-Rolla

Bid Beasts Security Review

Sep 25th 2025 - Oct 2nd 2025

Table of Contents

Whoami

I am Pavel (aka *kode-n-rolla*), an independent security researcher.

Driven by a passion for decentralization and security, I focus on strengthening blockchain ecosystems.

I believe that security is an ongoing process, and my work contributes to making systems more resilient and reliable.

Disclaimer

This report reflects a time-boxed security review performed as part of a ***CodeHawks*** audit contest.

The analysis focuses exclusively on the security aspects of the reviewed ***Solidity*** codebase and does not guarantee the discovery of all potential vulnerabilities.

Findings represent the issues identified during the allotted review period and should not be considered an endorsement of the underlying protocol, business model, or operational security practices.

Risk Classification

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

I used *CodeHawks* severity matrix to determine severity.

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

- *High Impact*:
    - Funds are directly or nearly directly at risk

    - There's a severe disruption of protocol functionality or availability.
- *Medium Impact*:
    - Funds are indirectly at risk.
    - There's some level of disruption to the protocol's functionality or availability.
- *Low Impact*:
    - Funds are not at risk.
    - However, a function might be incorrect, the state might not be handled appropriately, etc.

Protocol Summary

BidBeasts is an auction-based NFT marketplace implemented as a *Solidity* smart contract and designed for deployment on Ethereum-compatible networks. The protocol enables owners of *BidBeasts ERC721* tokens to list their NFTs for time-limited auctions, accept bids denominated in ETH, and finalize auctions with an integrated platform fee mechanism.

NFT owners can list tokens by transferring custody of the NFT to the marketplace contract and specifying a minimum acceptable price. During the auction period, participants may place bids by sending *ETH*, with each new bid required to exceed the current highest bid. The protocol automatically refunds the previous highest bidder when they are outbid.

Auctions are finalized after a fixed duration of three days and can be settled by any caller. If the highest bid meets or exceeds the minimum price, the NFT is transferred to the winning bidder and the seller receives the proceeds minus a fixed marketplace fee. If no valid bids are placed, the NFT is returned to the original seller.

The contract owner retains exclusive control over withdrawing accumulated platform fees. From a security perspective, the protocol relies on correct handling of NFT

custody, ETH transfers and refunds, auction state transitions, time-based logic, and privileged access to fee withdrawal functionality.

Audit Details

This audit was conducted as part of a time-boxed competitive security review hosted on the *CodeHawks* platform. The assessment focused exclusively on the ***Solidity*** smart contract logic and its adherence to intended game mechanics, security assumptions, and fund safety.

The review did not include off-chain components, frontend integrations, or economic modeling beyond what is explicitly enforced by the contract code.

The review was performed during the "Bid Beasts" *CodeHawks* audit event.

Scope

The audit scope consisted of the following smart contract:

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

Roles

- ***Seller*** **(NFT Owner)**

The Seller is an account that owns a BidBeasts ERC721 token and lists it for auction by transferring custody of the NFT to the marketplace contract. Upon successful auction completion, the Seller is entitled to receive the highest bid amount minus the platform fee.

- ***Bidder***

A Bidder is any account that participates in an active auction by submitting ETH bids. The highest Bidder at the end of the auction receives custody of the NFT if the auction is successfully settled.

- ***Contract Owner* (Platform Administrator)**

The Contract Owner is a privileged account responsible for deploying the marketplace contract and managing platform-level operations. This role is exclusively authorized to withdraw accumulated marketplace fees.

Executive Summary

This security review assessed the BidBeasts auction marketplace smart contract, with a focus on NFT custody, ETH fund flows, auction settlement logic, and privileged operations.

The review identified two high-severity issues that critically impact asset safety. The first issue allows arbitrary burning of BidBeasts NFTs, enabling any caller to permanently destroy NFTs held by the marketplace contract. This breaks core ownership guarantees and can lead to irreversible loss of user assets.

The second high-severity issue allows unauthorized withdrawal of ETH credited to other users as failed-transfer refunds. Due to insufficient access control over refund balances, an attacker can claim ETH that does not belong to them, resulting in direct theft of funds from other auction participants.

Both issues violate fundamental trust assumptions of the marketplace. Without remediation, users risk permanent loss of NFTs and ETH even when interacting with the protocol as intended. Collectively, these vulnerabilities render the marketplace unsafe for handling user assets in adversarial environments.

Issue found

Severity	Issue
High	Arbitrary NFT Burn Due to Missing Access Control
High	Arbitrary Withdrawal of Another Address' Failed-Transfer Credits

Findings

High

[H-01] Arbitrary NFT Burn Due to Missing Access Control

Description

In *BidBeasts_NFT_ERC721.sol*, the burn function is declared as public and directly calls the internal *_burn* function without performing any ownership or approval checks. As a result, any address can burn any existing NFT by providing a valid *tokenId*.

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

This allows unauthorized callers to permanently destroy NFTs they do not own.

Impact

- **Permanent Asset Loss:** Attackers can irreversibly destroy NFTs belonging to other users.
- **Denial of Service:** Large-scale burning can effectively destroy the entire NFT collection.

- **Loss of Trust**: The integrity of the NFT collection and the associated marketplace is fundamentally compromised.

Likelihood

- The function is publicly callable and requires only a valid *tokenId*.
- No ownership, approval, or role-based checks are enforced.
- The attack is trivial and does not require any special conditions.

Proof of Concept

A malicious user calls *burn(tokenId)* for any existing NFT they do not own. The call succeeds and permanently removes the token from the collection.

```
function test_anyoneCanBurnNFT() public {
    // Mint NFT to SELLER
    _mintNFT();
    assertEq(nft.ownerOf(TOKEN_ID), SELLER);

    // BIDDER_1 (not the owner) burns SELLER's NFT
    vm.prank(BIDDER_1);
    nft.burn(TOKEN_ID);

    // Token no longer exists
    vm.expectRevert();
    nft.ownerOf(TOKEN_ID);
}
```

Recommended Mitigation

Restrict access to the *burn* function by enforcing ownership or operator approval checks.

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

Follow the *OpenZeppelin* ERC721Burnable pattern by requiring that the caller is either the token owner or an approved operator.

[H-02] Arbitrary Withdrawal of Another Address' Failed-Transfer Credits

Description

In *withdrawAllFailedCredits(address receiver)*, the function **reads** the credit using *failedTransferCredits[receiver]* but **clears** a **different key** (e.g., *failedTransferCredits[msg.sender]*) and **pays** *msg.sender*

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

@>      failedTransferCredits[msg.sender] = 0; // <-- WRONG key cleared

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

Because the credit entry for receiver is never cleared, an attacker can repeatedly call *withdrawAllFailedCredits(receiver)* and receive the same amount each time, until the contract's ETH balance is exhausted.

Root Cause

A **mapping key mismatch** between read, clear/write, and pay operations:

- **Read:** *failedTransferCredits[receiver]*
- **Clear/Write:** *failedTransferCredits[msg.sender]* (or another mismatched key)
- **Pay:** *msg.sender*

This logic bug leaves the victim's credit intact while transferring funds to the attacker, enabling repeatable withdrawals.

Impact

- **Direct Theft of Funds:** The same credited amount can be withdrawn repeatedly.
- **Pool Drain:** Theft continues until the contract balance is depleted.
- **User & Protocol Loss:** Seller refunds and user credits can be stolen, causing financial loss and reputational damage.

Likelihood

- Requires only a nonzero *failedTransferCredits[receiver]* entry.
- The function is publicly callable; no privileges are required.
- An attacker can **self-bootstrap** the exploit by causing refunds to fail (e.g., bidding from a contract that rejects ETH), then repeatedly withdrawing.

Proof of Concept

A test demonstrates that calling *withdrawAllFailedCredits(receiver)* repeatedly transfers ETH to the attacker while the credit entry for receiver remains unchanged.

```
function test_withdrawOtherFunds_PoC() public {
    // 1) setup: mint & list NFT
    _mintNFT();
    _listNFT();

    // 2) prepare a contract that cannot receive ETH
    // RejectEther has no payable receive/fallback
    vm.deal(address(rejector), STARTING_BALANCE);

    // 3) make first bid from rejector (non-payable recipient)
    uint256 bidToSteal = MIN_PRICE + 1;
    vm.prank(address(rejector));
    market.placeBid{value: bidToSteal}(TOKEN_ID);

    // 4) make a higher bid from another account to trigger refund to rejector
    vm.prank(BIDDER_1);
    market.placeBid{value: 3 ether}(TOKEN_ID);

    // now failedTransferCredits[rejector] == bidToSteal
    assertEq(market.failedTransferCredits(address(rejector)), bidToSteal);

    // 5) attacker withdraws other user's credits (bug)
```

```
    uint256 attackerBefore = BIDDER_2.balance;
    vm.prank(BIDDER_2);
    market.withdrawAllFailedCredits(address(rejector));

    // attacker received the credit
    assertEq(BIDDER_2.balance, attackerBefore + bidToSteal);

    // victim's mapping was NOT cleared -> attacker can repeat
    assertEq(market.failedTransferCredits(address(rejector)), bidToSteal);

    // repeat attack (still succeeds in buggy version while contract has funds)
    vm.prank(BIDDER_2);
    market.withdrawAllFailedCredits(address(rejector));
    assertEq(BIDDER_2.balance, attackerBefore + bidToSteal * 2);
}
```

Recommended Mitigation

- **Bind read, clear, and pay to the same key**, and restrict withdrawals so **only the credited address** can withdraw its own funds.
- Clear the credit **before** transferring ETH (checks-effects-interactions).

```
- function withdrawAllFailedCredits(address _receiver) external {
+ function withdrawAllFailedCredits(address _receiver) external nonReentrant {
+     require(_receiver == msg.sender, "Only receiver can withdraw");
      uint256 amount = failedTransferCredits[_receiver];
      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");
}
```

Why this works:

Using the same key for read/clear/pay ensures credits are consumed atomically and only by their rightful owner.

Additional Protections (Recommended)

- Add *nonReentrant* (*OpenZeppelin ReentrancyGuard*) to:
  - *withdrawAllFailedCredits*
  - other ETH-moving functions such as *placeBid*, *takeHighestBid*, *settleAuction*, *withdrawFee*