e

CD SECURITY

AUDIT REPORT

Midnight Second Review

October 2024

Prepared by
Varun_05
Pelz

Introduction

A time-boxed security review of the **Midnight** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Midnight**

The protocol implements a classic NFT contract with the addition of generations and traits. Only the first generation can be minted. All other generations are minted only via breeding of previous generations.

Owners have the ability to put their NFTs for rent so other users can use them to breed. The protocol also has a marketplace where users can list their NFTs for sale, auction them off, bid or create offers to existing listings.

Lastly, the protocol offers users the ability to participate in a Chainlink VRF backed raffle where they can win a chance to buy an NFTs in the initial pre-sale.

Severity classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 17d393db121b02762fe9d56579491589f26a50c2**

Scope

The following smart contracts were in scope of the audit:

- `GHOSTSBREEDING.sol`
- `GHOSTSRAFFLE.sol`
- `GeneScience.sol`
- `MIDNIGHT.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 17 issues
- Medium: 4 issues
- Low: 2 issues

Findings Summary

ID	Title	Severity
[C-01]	Wrong check used in `requestDelivery` for auctioned NFTs	Critical
[C-02]	Production fee for listed NFT is not accounted in `requestProduction`	Critical
[C-03]	Arbitrary NFT address can be passed by the buyer and lead to problems	Critical
[C-04]	Wrong nft can be sent to the offerer by the seller of a listed nft	Critical
[C-05]	A renter can cause the DOS of rent functionality for a particular nft	Critical
[C-06]	`requestHouseAuthentication` Fails Due to Missing onERC721Received	Critical
[H-01]	Wrong check for physical type NFTs in requestProduction function	High
[H-02]	Malicious auction creator can set `_tokenId = true` for wrong `tokenId`	High
[H-03]	Wrong check for setting/updating gene science address	High
[H-04]	Wrong renter check used in `breedWithAuto()`	High
[H-05]	User can breed with rented NFT even after the rental end time	High
[H-06]	A user can even rent the NFT without paying any fees	High
[H-07]	An offerer can get back more than offered price	High
[H-08]	Wrong NFT Approval Check Allows Unauthorized NFT Listing	High
[H-09]	Unauthorized Request for NFT Delivery	High
[H-10]	Unlimited NFT Delivery Requests Leading to Loss of Funds	High
[H-11]	Auction NFT Theft via Missing NFT Address Validation	High
[M-01]	Minimum bid check is not correctly implemented	Medium

ID	Title	Severity
[M-02]	User can avoid using `breedWithRented` by directly calling `breedWithAuto`	Medium
[M-03]	There is no check for breeding starting date for the sire generation	Medium
[M-04]	`sireAllowedToAddress[_sireId]` is not allowed to breed	Medium
[L-01]	`getAllAuctions` function uses incorrect `auctionNfts` mapping	Low
[L-02]	Predictable Randomness in Winner Selection	Low

Detailed Findings

[C-01] Wrong check used in `requestDelivery` for auctioned NFTs

Severity

Impact: High

Likelihood: High

Description

In order to request delivery of the NFT it should be of physical type whereas in the current implementation to check if the auctioned NFT is physical or not wrong check is used which checks if the auctioned NFT is not a physical type which is wrong and it won't allow request delivery of the physical NFTs.

```
function requestDelivery(
      uint256 _listId,
      uint256 _auctionId,
      bool isAuctionNft
    ) external payable {
      if(isAuctionNft){
          AuctionNFT storage auction = auctionNfts[_auctionId];
===>      require(!auction.isPhysical , "Physical item required for
authentication");
      // Ensure the fee is paid
      require(msg.value >= auction.deliveryFee, "Invalid Fee");
      // Ensure the sale has already been confirmed
      require(auction.status == Status.COMPLETED,"Item not sold yet");

      // update NFT Delivery status
      auction.deliveryStatus = DeliveryStatus.REQUESTED;

      (bool sent, ) = payable(auction.creator).call{
        value: msg.value
        }("");
      require(sent, "Ghost: Failed to transfer fee to Seller.");
      emit Deliveryrequest(
        _listId,
        msg.value
       );
      ....... Rest of code
```

Recommendations

Mitigation is simple just add the following:

```

function requestDelivery(
      uint256 _listId,
      uint256 _auctionId,
      bool isAuctionNft
    ) external payable {
      if(isAuctionNft){
          AuctionNFT storage auction = auctionNfts[_auctionId];
--  require(!auction.isPhysical , "Physical item required for
authentication");
++    require(auction.isPhysical , "Physical item required for
authentication");
      // Ensure the fee is paid
      require(msg.value >= auction.deliveryFee, "Invalid Fee");
      // Ensure the sale has already been confirmed
      require(auction.status == Status.COMPLETED,"Item not sold yet");

      // update NFT Delivery status
      auction.deliveryStatus = DeliveryStatus.REQUESTED;

      (bool sent, ) = payable(auction.creator).call{
        value: msg.value
        }("");
      require(sent, "Ghost: Failed to transfer fee to Seller.");
      emit Deliveryrequest(
        _listId,
        msg.value
       );
      ...... Rest of Code
```

[C-02] Production fee for listed NFT is not accounted in `requestProduction`

Severity

Impact: High

Likelihood: High

Description

There is a minimum amount that should be paid to the seller of the listed NFT for production and delivery of the physical NFT. In the current implementation production fee is not accounted in production of sold listed NFTs as can be seen from the below code whereas it is correctly accounted for the auctioned NFTs.

```
 function requestProduction(
      uint256 _listId,
      uint256 _auctionId,
      bool isAuctionNft
    ) external payable {
      if(isAuctionNft){
      AuctionNFT storage auction = auctionNfts[_auctionId];
      require(!auction.isPhysical , "Physical item required for
authentication");
      // Ensure the fee is paid
===>      require(msg.value >= auction.deliveryFee + productionFee,
"Invalid Fee");
      // Ensure the sale has already been confirmed
      require(auction.status == Status.COMPLETED,"Item not sold yet");

      // update NFT Delivery status
      auction.deliveryStatus = DeliveryStatus.REQUESTED;

      (bool sent, ) = payable(auction.creator).call{
        value: msg.value
        }("");
      require(sent, "Ghost: Failed to transfer fee to Seller.");
      emit Deliveryrequest(
        _listId,
        msg.value
       );

      } else {

      ListNFT storage listedNft = listNfts[_listId];

      // Ensure the caller is the owner of the NFT
      // Ensure the NFT is physical
      require(!listedNft.isPhysical , "Physical item required for
authentication");
      // Ensure the fee is paid
 ===>     require(msg.value >= listedNft.deliveryFee, "Invalid Fee");
      // Ensure the sale has already been confirmed
      require(listedNft.status == Status.COMPLETED,"Item not sold yet");

      // update NFT Delivery status
      listedNft.deliveryStatus = DeliveryStatus.REQUESTED;

      (bool sent, ) = payable(listedNft.seller).call{
        value: msg.value
        }("");
      require(sent, "Ghost: Failed to transfer fee to Seller.");
      emit Deliveryrequest(
        _listId,
        msg.value
       );
      }

    }
```

Recommendations

Modify the code as follows:

```
 function requestProduction(
      uint256 _listId,
      uint256 _auctionId,
      bool isAuctionNft
    ) external payable {
     ...
      // Ensure the caller is the owner of the NFT
      // Ensure the NFT is physical
      require(!listedNft.isPhysical , "Physical item required for
authentication");
      // Ensure the fee is paid
--     require(msg.value >= listedNft.deliveryFee, "Invalid Fee");
++    require(msg.value >= listedNft.deliveryFee + productionFee, "Invalid
Fee");
      // Ensure the sale has already been confirmed
      require(listedNft.status == Status.COMPLETED,"Item not sold yet");

      // update NFT Delivery status
      listedNft.deliveryStatus = DeliveryStatus.REQUESTED;

      (bool sent, ) = payable(listedNft.seller).call{
        value: msg.value
        }("");
      require(sent, "Ghost: Failed to transfer fee to Seller.");
      emit Deliveryrequest(
        _listId,
        msg.value
       );
      }

    }
```

[C-03] Arbitrary NFT address can be passed by the buyer and lead to problems

Severity

Impact: High

Likelihood: High

Description

Following is the buy NFT function:

```
 function buyNFT(
    address _nft,
    uint256 _listId
    ) external nonReentrant isListedNFT(_listId) payable {
    ListNFT storage listedNft = listNfts[_listId];
    // require(_tokenId > 10, "NFT is already sold through Raffle");
    require(!listedNft.sold, "NFT already sold");
    require( msg.value == listedNft.price, "Invalid price");
    //Audit fix [H-09]
    require(listedNft.status !=Status.CANCELLED, " cannot buy the
cancelled nft");
    listedNft.sold = true;
    listedNft.status = Status.COMPLETED;
    uint256 totalPrice = msg.value;


    if (!nftSold[_nft][listedNft.tokenId]) {
        _processInitialSale(totalPrice,_nft);
        nftSold[_nft][listedNft.tokenId] = true;
    } else {
        totalPrice = _processResale(totalPrice,_nft);
        (bool sent, ) = payable(listedNft.seller).call{
        value: totalPrice
        }("");
        require(sent, "Ghost: Failed to transfer fee to fee to
MidNight.");
    }




    IERC721(listedNft.nft).safeTransferFrom(
        address(this),
        msg.sender,
        listedNft.tokenId
    );

    emit BoughtNFT(
        listedNft.nft,
        listedNft.tokenId,
        msg.value,
        listedNft.seller,
        msg.sender
```

```
      );
    }
```

As can be seen that the _nft address is passed by the buyer of the NFT. Now lets see what issue can occur if _nft address other than the listed.nft is passed bu the buyer.

1. Currently ghosts nfts are only allowed to be listed and once they are listed then that means that they are resaled because ghosts NFTs can be acquired through auction or through raffle system. So when the resaled NFTs are bought by the buyer and the buyer mentions _nft address as arbitrary one lets assume he passes address 0 then what happens is following code is executed:

```
totalPrice = _processResale(totalPrice,_nft);
        (bool sent, ) = payable(listedNft.seller).call{
        value: totalPrice
        }("");
        require(sent, "Ghost: Failed to transfer fee to fee to
MidNight.");
```

Now _processResale(totalPrice,0) will be called _processResale function is as follows

```
 function _processResale(
    uint256 totalPrice,
    address nft
    ) internal returns (uint256) {
    uint256 totalShareAfterMidnightShare = ((totalPrice *
midNightAndProjectFeeShare) / 1000) - ((totalPrice *
reSaleMarketplaceMidNightFee) / 1000);

    (bool sent, ) = midNightAddress.call{
        value: ((totalPrice * reSaleMarketplaceMidNightFee) / 1000)
        }("");
    require(sent, "Ghost: Failed to transfer fee to fee to MidNight.");

    uint256 charityRoyaltyShare = (totalShareAfterMidnightShare * 450) /
1000;  // 45% of 4.5% for charityWallet
    uint256 projectRoyaltyShare = (totalShareAfterMidnightShare * 550) /
1000; // 55% of 4.5% for projectRoyalty



    // Transfer to Charity Treasury Wallet
    (bool transfer, ) = charityTreasuryWallet[nft].call{
        value: charityRoyaltyShare
        }("");
    require(transfer, "Ghost: Failed to transfer fee to fee to
charityTreasuryWallet.");

    // Transfer remaining amount to Project Wallet
    (bool response, ) = projectWallet[nft].call{
        value: projectRoyaltyShare
        }("");
    require(response, "Ghost: Failed to transfer fee to fee to
projectWallet.");

    return totalPrice -((totalPrice * midNightAndProjectFeeShare) / 1000);
    }
```

As address 0 is passed as NFT so the charity royalty share and project royalty share are sent to other unintended address. Note that the _nft address passed by the buyer can be anything not necessarily address. It can also be any other NFTs wallet that would be added to the midnight marketplace

2. If the NFT listed is resaled but the buyer passes arbitrary _nft address then it would behave as if the NFT is sold first time and the seller of the NFT would not get any money for the resaled NFT therefore loss of money for the resaler.

Recommendations

Don't take _nft as input use `listedNft.nft` instead everywhere.

[C-04] Wrong nft can be sent to the offerer by the seller of a listed NFT

Severity

Impact: High

Likelihood: High

Description

Wrong NFT can be sent to the offerer by the seller of the listed NFT . This can happens as follows.

1. The offerer sends the offered price to the contract for a particular listed NFT which is as follows.

```
function offerNFT(
        address _nft,
        uint256 _listId,
        uint256 _tokenId
    ) external isListedNFT(_listId) nonReentrant payable {
        require(msg.value > 0, "price can not 0");

         //Audit Concern [H-07]
        OfferNFT storage offer = offerNfts[_nft][_tokenId][msg.sender];
        // require(offer.offerer != msg.sender,"Already offer by this
user");
        if(offer.offerer == msg.sender){
        (bool sent, ) = payable(offer.offerer).call{
        value: offer.offerPrice
        }("");
      require(sent, "Ghost: Failed to transfer fee to Seller.");
      }
        ListNFT memory nft = listNfts[_listId];
        offerNfts[_nft][nft.tokenId][msg.sender] = OfferNFT({
            nft: nft.nft,
            tokenId: nft.tokenId,
            offerer: msg.sender,
            offerPrice: msg.value,
            accepted: false,
            listingId: _listId
        });



        emit OfferredNFT(
            nft.nft,
            nft.tokenId,
            msg.value,
            msg.sender
        );
    }
```

Now the correct flow is that the seller of the listed supply for which the offerer has sent the offerer price can accept it or not. However, the issue is that any seller of any listed NFT can accept the offerer's price and send the NFT to the offerer. The issue is that the offerer would receive the wrong NFT in this case. This can happen because when the seller of any NFT finds that a better offer for a particular NFT is available then they can call accept offer function. Following is the accept offer function:

```
function acceptOfferNFT(
    address _nft,
    uint256 _tokenId,
    address _offerer,
    uint256 _listingId,
    uint256 _price
) external nonReentrant isOfferredNFT(_nft, _tokenId, _offerer)
isListedNFT(_listingId) {
    require(listNfts[_listingId].seller == msg.sender, "Not listed
owner");

    OfferNFT storage offer = offerNfts[_nft][_tokenId][_offerer];
    ListNFT storage list = listNfts[_listingId];
    require(_price == offer.offerPrice,"invalid price");
    require(!list.sold, "Already sold");
    require(!offer.accepted, "Offer already accepted");
    // //Audit Concern [H-06]
    // require(offer.listingId==list.listingId, " listing and offer id
mismatch");

    list.sold = true;
    list.status = Status.COMPLETED;
    offer.accepted = true;
    uint256 offerPrice = offer.offerPrice;


    if (!nftSold[_nft][_tokenId]) {
        _processInitialSale(offerPrice,_nft);
        nftSold[_nft][_tokenId] = true;
    } else {
        offerPrice = _processResale(offerPrice,_nft);
        (bool success, ) = payable(list.seller).call{value: offerPrice}
("");
        require(success, "Transfer to seller failed");
    }


      //transfer nft to offerer
    IERC721(list.nft).safeTransferFrom(
        address(this),
        offer.offerer,
        list.tokenId
    );

    emit AcceptedNFT(
        offer.nft,
        offer.tokenId,
        offer.offerPrice,
        offer.offerer,
        list.seller
    );
}
```

Now as we can see it is only checked that NFT is offered by the `isOfferredNFT(\_nft, \_tokenId, \_offerer)` modifier and whether the NFT is listed by `isListedNFT(\_listingId)` modifier. It is not checked that the listing id of the offer matches the listing id of the NFT whose seller is accepting the offer. Due to this wrong NFT will be sent to the offerer.

Recommendations

Add the following line in the accept offer function

```
 require(offer.listingId==list.listingId, " listing and offer id
mismatch");
```

[C-05] A renter can cause the DOS of rent functionality for a particular NFT

Severity

Impact: High

Likelihood: HIgh

Description

Following is returnRentedNft

```
 function returnRentedNFT(uint256 _nftId) public nonReentrant{
        Rental storage rental = rentals[_nftId];
        require(rental.renter == msg.sender, "Not the renter");
        rental.renter = address(0);
        rental.rentalEndTime = 0;
        rental.isRented = false;
        emit NFTReturned(_nftId, msg.sender);
    }
```

As from above we can see that only the current renter of the NFT can call this function to return the rented NFT . So now a renter can just not call this function ever thus this NFT can't be again used for renting purpose ever. Even after the rental end time the NFT would still be with the renter. Thus causing DOS of rent functionality

Recommendations

Allow `returnRentedNft` to be called by anyone when the block.timestamp is greater than rental End time.

[C-06] `requestHouseAuthentication` Fails Due to Missing onERC721Received Callback

Severity

Impact: High

Likelihood: High

Description

In the `MIDNIGHT.sol` contract, the `requestHouseAuthentication` function is responsible for requesting authentication of an NFT. However, the function will always fail due to the use of the `safeTransferFrom` function on line 600:

```
IERC721(_nftAddress).safeTransferFrom(
    msg.sender,
    address(this),
    _tokenId
); // @audit-issue no request will be successful!!
```

The contract does not implement the onERC721Received callback, which is required when transferring NFTs using safeTransferFrom. As a result, the transfer will revert, and no authentication request will be successful. This prevents users from completing the authentication process for their NFTs.

Recommendations

To resolve this issue, implement the onERC721Received function to handle the receipt of NFTs. This callback is required by the ERC-721 standard to ensure the safe transfer of tokens to contracts.

```
function onERC721Received(
    address operator,
    address from,
    uint256 tokenId,
    bytes calldata data
) external override returns (bytes4) {
    return this.onERC721Received.selector;
}
```

By implementing this callback, the contract will be able to handle NFT transfers successfully, allowing users to request house authentication without any failures.

[H-01] Wrong check for physical type NFTs in requestProduction()

Severity

Impact: Medium

Likelihood: High

Description

In order to request production of the NFT it should be of physical type whereas in the current implementation to check if the NFT is physical or not wrong check is used which checks if the NFT is not a physical type which is wrong and it won't allow request production of the physical NFTs.

```
function requestProduction(
      uint256 _listId,
      uint256 _auctionId,
      bool isAuctionNft
    ) external payable {
      if(isAuctionNft){
      AuctionNFT storage auction = auctionNfts[_auctionId];
 ===>     require(!auction.isPhysical , "Physical item required for
authentication");
      // Ensure the fee is paid
      require(msg.value >= auction.deliveryFee + productionFee, "Invalid
Fee");
      // Ensure the sale has already been confirmed
      require(auction.status == Status.COMPLETED,"Item not sold yet");

      // update NFT Delivery status
      auction.deliveryStatus = DeliveryStatus.REQUESTED;

      (bool sent, ) = payable(auction.creator).call{
        value: msg.value
        }("");
      require(sent, "Ghost: Failed to transfer fee to Seller.");
      emit Deliveryrequest(
        _listId,
        msg.value
       );

      } else {

      ListNFT storage listedNft = listNfts[_listId];

      // Ensure the caller is the owner of the NFT
      // Ensure the NFT is physical
 ===>     require(!listedNft.isPhysical , "Physical item required for
authentication");
      // Ensure the fee is paid
      require(msg.value >= listedNft.deliveryFee, "Invalid Fee");
      // Ensure the sale has already been confirmed
      require(listedNft.status == Status.COMPLETED,"Item not sold yet");

      // update NFT Delivery status
      listedNft.deliveryStatus = DeliveryStatus.REQUESTED;

      (bool sent, ) = payable(listedNft.seller).call{
        value: msg.value
        }("");
      require(sent, "Ghost: Failed to transfer fee to Seller.");
      emit Deliveryrequest(
        _listId,
        msg.value
       );
      }

    }
```

Recommendations

Modify the code as follows:

```
function requestProduction(
      uint256 _listId,
      uint256 _auctionId,
      bool isAuctionNft
    ) external payable {
      if(isAuctionNft){
      AuctionNFT storage auction = auctionNfts[_auctionId];
--     require(!auction.isPhysical , "Physical item required for
authentication");
++   require(auction.isPhysical , "Physical item required for
authentication");
      // Ensure the fee is paid
      require(msg.value >= auction.deliveryFee + productionFee, "Invalid
Fee");
      // Ensure the sale has already been confirmed
      require(auction.status == Status.COMPLETED,"Item not sold yet");

      // update NFT Delivery status
      auction.deliveryStatus = DeliveryStatus.REQUESTED;

      (bool sent, ) = payable(auction.creator).call{
        value: msg.value
        }("");
      require(sent, "Ghost: Failed to transfer fee to Seller.");
      emit Deliveryrequest(
        _listId,
        msg.value
       );

      } else {

      ListNFT storage listedNft = listNfts[_listId];

      // Ensure the caller is the owner of the NFT
      // Ensure the NFT is physical
--    require(!listedNft.isPhysical , "Physical item required for
authentication");
++      require(listedNft.isPhysical , "Physical item required for
authentication");
      // Ensure the fee is paid
      require(msg.value >= listedNft.deliveryFee, "Invalid Fee");
      // Ensure the sale has already been confirmed
      require(listedNft.status == Status.COMPLETED,"Item not sold yet");

      // update NFT Delivery status
      listedNft.deliveryStatus = DeliveryStatus.REQUESTED;

      (bool sent, ) = payable(listedNft.seller).call{
        value: msg.value
        }("");
      require(sent, "Ghost: Failed to transfer fee to Seller.");
      emit Deliveryrequest(
        _listId,
        msg.value
       );
      }
    }
```

[H-02] Malicious auction creator can set `_tokenId = true` for wrong `tokenId`

Severity

Impact: High

Likelihood: Medium

Description

When the auction concludes the result auction function can be called by either creator, the winner of the auction, or the owner of the midnight marketplace. Now suppose the owner of a NFT calls the result function with the tokenId for a particular NFT other than the tokenId for which the auction was held. Following if and else conditions are of the main focus:

```
if (!nftSold[_nft][_tokenId]) {
        // If it's an initial sale
        _processInitialSale(totalPrice,_nft);
        nftSold[_nft][_tokenId] = true;
    } else {
        // If it's a resale, process royalty
        totalPrice = _processResale(totalPrice,_nft);
        (bool success, ) = payable(auction.creator).call{value:
totalPrice}("");
        require(success, "Transfer to auction creator failed");
    }
```

Now assuming that it's an initial sale which can very likely happen. Now as incorrect tokenId is used then nftSold[_nft][_tokenId] = true will be set for NFT.

```
 nft.transferFrom(address(this), auction.lastBidder, auction.tokenId);
```

From the above code we can see that NFT with auction.tokenId is sent to the winner of the auction. As malicious creator had passed `tokenId` different from `auction.tokenId`.

Now suppose the winner of the auction decides to resale the NFT that he won the bid for by listing it. Issue is when someone buys the NFT then follows if condition will be executed

```
function buyNFT(
    address _nft,
    uint256 _listId
    ) external nonReentrant isListedNFT(_listId) payable {
    ListNFT storage listedNft = listNfts[_listId];
    // require(_tokenId > 10, "NFT is already sold through Raffle");
    require(!listedNft.sold, "NFT already sold");
    require( msg.value == listedNft.price, "Invalid price");
    //Audit fix [H-09]
    require(listedNft.status !=Status.CANCELLED, " cannot buy the
cancelled nft");
    listedNft.sold = true;
    listedNft.status = Status.COMPLETED;
    uint256 totalPrice = msg.value;


    if (!nftSold[_nft][listedNft.tokenId]) {
        _processInitialSale(totalPrice,_nft);
        nftSold[_nft][listedNft.tokenId] = true;
    } else {
        totalPrice = _processResale(totalPrice,_nft);
        (bool sent, ) = payable(listedNft.seller).call{
        value: totalPrice
        }("");
        require(sent, "Ghost: Failed to transfer fee to fee to
MidNight.");
    }
```

Because the malicious creator had set nftSold[_nft][listedNft.tokenId] = true for different tokenId, due to this

```
 if (!nftSold[_nft][listedNft.tokenId]) {
        _processInitialSale(totalPrice,_nft);
        nftSold[_nft][listedNft.tokenId] = true;
```

The above condition would be executed. The effect of this is that the seller who resaled the NFT wouldn't get any price for selling the NFT and all the money would be used up as royalties.

Recommendations

Implement the following check in result auction function:

```
 require(auction.tokenId == _tokenId , "Invalid NFT Address");
```

[H-03] Wrong check used for setting/updating gene science address

Severity

Impact: High

Likelihood: Medium

Description

Following is `setGeneScienceAddress()`:

```
 function setGeneScienceAddress(address _address) external onlyOwner {
        GeneScienceInterface candidateContract =
GeneScienceInterface(_address);
        require(candidateContract.mixGenes(0, 0, 0) != 0, "Invalid gene
science contract");
        geneScience = candidateContract;
    }
```

Now the following check is wrongly used:

```
 require(candidateContract.mixGenes(0, 0, 0) != 0, "Invalid gene science
contract");
```

This is because mixGenes(0,0,0) will always return 0 because mixing two zero genes would always return 0.

Recommendations

Change the check as follows:

```
 require(candidateContract.mixGenes(0, 0, 0) = 0, "Invalid gene science
contract");
```

[H-04] Wrong renter check used in breedWithAuto function

Severity

Impact: High

Likelihood: Medium

Description

Following is `breedWithAuto` function:

```
function breedWithAuto(uint256 _matronId, uint256 _sireId) public
nonReentrant {
        NFT storage matron = nfts[_matronId];
        //Added
        require((ownerOf(_sireId) == msg.sender &&
        rentals[_matronId].renter == msg.sender) ||
        (ownerOf(_matronId) == msg.sender &&
        ownerOf(_sireId) == msg.sender), "Not owner nor renter");
        require(_isReadyToBreed(matron), "Matron not ready to breed");
        NFT storage sire = nfts[_sireId];
        require(_isReadyToBreed(sire), "Sire not ready to breed");
        require(_isValidMatingPair(matron, _matronId, sire, _sireId),
"Invalid mating pair");
        //Breeding cannot start before the defined date for the generation
        require(block.timestamp >=
generationBreedingStartDate[matron.generation], "Breeding has not started
for this generation");
        //Breeding cannot exceed the supply limit for the generation
        require(generationBreedingSupply[matron.generation] > 0, "Breeding
supply exhausted for this generation");
        breedWith(_matronId, _sireId);
        //decrementing generation breeding supply
        generationBreedingSupply[matron.generation]--;
    }
```

In this issue we will look at the following check:

```
 require((ownerOf(_sireId) == msg.sender &&
        rentals[_matronId].renter == msg.sender) ||
        (ownerOf(_matronId) == msg.sender &&
        ownerOf(_sireId) == msg.sender), "Not owner nor renter");
```

As we can see that the above checks that either the msg.sender should be the owner of both sire id and matron id or he should rent the matron id but it is wrong because suppose you rent matronId. Now you would be paying the rent according to number of days you are renting the matron id for . Rent function is as follows:

```
function rentNFT(uint256 _nftId, uint256 NumOfDays) external payable
nonReentrant{
        Rental storage rental = rentals[_nftId];
        require(NumOfDays<=rental.maxDaysforrent," Days mismatch");
        require(!rental.isRented, "NFT already rented");
        require(block.timestamp < rental.rentalEndTime, "Rental period
expired");
        require(msg.value == (rental.rentalFeePerday * NumOfDays),
"Insufficient rental fee");
        (bool sent, ) = ownerOf(_nftId).call{value: msg.value}("");
        require(sent, "Failed to transfer fee to the owner of nft.");
        rental.renter = msg.sender;
        rental.rentalEndTime = block.timestamp +  (NumOfDays *
secondsPerDay);
        rental.tokenID = _nftId;
        rental.isRented = true;
        emit NFTRented(_nftId, msg.sender, rental.rentalEndTime);
    }
```

As from above we can see we pay rent for the matron id. Now suppose the matron is ready to give birth then the following function is called:

```
function giveBirth(uint256 _matronId) external nonReentrant returns
(uint256) {
    NFT storage matron = nfts[_matronId];
    require(_isReadyToGiveBirth(matron), "Not ready to give birth");
    uint256 sireId = matron.siringWithId;
    NFT storage sire = nfts[sireId];
        isBred[_matronId] = true;
        isBred[sireId] = true;
    uint256 parentGen = matron.generation;
    if (sire.generation > matron.generation) {
        parentGen = sire.generation;
    }
    // Mix genes using the GeneScience contract
    uint256 childGenes = geneScience.mixGenes(matron.genes, sire.genes,
matron.cooldownEndTime - 1);
    // check the owner of breed nft either its is a rented nft or ownes by
user
    address owner = ownerOf(_matronId);
    // Update the generation and increment the supply for the new NFT
    uint256 childGeneration = parentGen + 1;
    uint256 child_nft_id=nfts.length;
    //uint256 newNftId = length;
    nfts.push(NFT(childGenes, 1, 0, childGeneration, block.timestamp));
    //push the bred date to mapping
       breedingDates[child_nft_id] = block.timestamp;
    // Increment the generation supply
    generationBreedingSupply[childGeneration]++;

    _safeMint(owner, child_nft_id);
    // Reset the matron's siring status
    delete matron.siringWithId;
    pregnantNFTs--;
    return child_nft_id;
    }
```

As we can see from the following lines that the new NFT is minted to the owner of the matron id and as we the user rented the matron id he would not be minted any NFT plus he gave rent for the matron id, so makes no sense that the user would rent matron id:

```
address owner = ownerOf(_matronId);
_safeMint(owner, child_nft_id);
```

So it doesn't makes sense to check that the user would ever rent matron. Instead user will always rent sire id but it won't pass the initial check which allows `msg.sender` to be the renter of only matron id and not sire id. Therefore wrong check implemented.

Recommendations

Change the check as follows:

```
 require((ownerOf(_matronId) == msg.sender &&
        rentals[_sireId].renter == msg.sender) ||
        (ownerOf(_matronId) == msg.sender &&
        ownerOf(_sireId) == msg.sender), "Not owner nor renter");
```

[H-05] User can breed with rented NFT even after the rental end time

Severity

Impact: High

Likelihood: Medium

Description

A user pays rent proportional to the number of days he rents the NFT and accordingly the rental end time is decided. So after the rental date the renter should not be able to use the rented NFT but currently there is no check introduced which disallows using the rented NFT after the rental end time. Due to this the renter can use the rented NFT for breeding even after the rental end time.

Recommendations

Introduce a check to prevent using the rented NFT after the rental end time in breedWithAuto function as breeding only happens through this function.

[H-06] A user can even rent the NFT without paying any fees

Severity

Impact: Medium

Likelihood: High

Description

Following is `rentNFT` function:

```
function rentNFT(uint256 _nftId, uint256 NumOfDays) external payable
nonReentrant{
        Rental storage rental = rentals[_nftId];
        require(NumOfDays<=rental.maxDaysforrent," Days mismatch");
        require(!rental.isRented, "NFT already rented");
        require(block.timestamp < rental.rentalEndTime, "Rental period
expired");
        require(msg.value == (rental.rentalFeePerday * NumOfDays),
"Insufficient rental fee");
        (bool sent, ) = ownerOf(_nftId).call{value: msg.value}("");
        require(sent, "Failed to transfer fee to the owner of nft.");
        rental.renter = msg.sender;
        rental.rentalEndTime = block.timestamp +  (NumOfDays *
secondsPerDay);
        rental.tokenID = _nftId;
        rental.isRented = true;
        emit NFTRented(_nftId, msg.sender, rental.rentalEndTime);
    }
```

Now from above as we can see that there is no check which forces users to not mention 0 days therefore paying no fees plus successfully able to rent the NFT. This finding is different from my previous one in which there was no check for rental end time because user can call rent NFT for 0 days and `breedWithAuto` in the same block and can use the rented functionality for free in order to breed.

Recommendations

Introduce a check to not allow 0 days to be passed.

[H-07] An offerer can get back more than offered price

Severity

Impact: High

Likelihood: Medium

Description

Following is `offerNft` function:

```
function offerNFT(
        address _nft,
        uint256 _listId,
        uint256 _tokenId
    ) external isListedNFT(_listId) nonReentrant payable {
        require(msg.value > 0, "price can not 0");

         //Audit Concern [H-07]
        OfferNFT storage offer = offerNfts[_nft][_tokenId][msg.sender];
        // require(offer.offerer != msg.sender,"Already offer by this
user");
        if(offer.offerer == msg.sender){
        (bool sent, ) = payable(offer.offerer).call{
        value: offer.offerPrice
        }("");
      require(sent, "Ghost: Failed to transfer fee to Seller.");
      }
        ListNFT memory nft = listNfts[_listId];
        offerNfts[_nft][nft.tokenId][msg.sender] = OfferNFT({
            nft: nft.nft,
            tokenId: nft.tokenId,
            offerer: msg.sender,
            offerPrice: msg.value,
            accepted: false,
            listingId: _listId
        });



        emit OfferredNFT(
            nft.nft,
            nft.tokenId,
            msg.value,
            msg.sender
        );
    }
```

Now suppose an offerer has already offered for a tokenId then according to the code he would be refunded the offered price if he changes the offer price for the same tokenId but suppose the user places an offer for different tokenId then what the code will do is refund the user the offered price for tokenId A and list a new offer for tokenId B . Now the offerer has got back his refund for tokenId A but the offerNfts[_nft][_tokenId A][msg.sender] has not been cleared so the offerer calls the cancel offer NFT function:

```
function cancelOfferNFT(address _nft, uint256 _tokenId)
        external
        nonReentrant
        isOfferredNFT(_nft, _tokenId, msg.sender)
    {
        OfferNFT memory offer = offerNfts[_nft][_tokenId][msg.sender];
        require(offer.offerer == msg.sender, "not offerer");
        require(!offer.accepted, "offer already accepted");
        delete offerNfts[_nft][_tokenId][msg.sender];
        (bool success, ) = payable(offer.offerer).call{value:
```

```
offer.offerPrice}("");
        require(success, "Transfer failed");
        emit CanceledOfferredNFT(
            offer.nft,
            offer.tokenId,
            offer.offerPrice,
            msg.sender
        );
    }
```

As can be seen from above now the offerer again gets back the offer price back thus getting double refund back. Note that the offerer can do this before his offer is accepted which is very likely.

Recommendations

Check the same tokenId is used as follows:

```
function offerNFT(
        address _nft,
        uint256 _listId,
        uint256 _tokenId
    ) external isListedNFT(_listId) nonReentrant payable {
        require(msg.value > 0, "price can not 0");
++        require(_tokenId == listNfts[_listId].tokenId," Wrong tokenId
used");
         //Audit Concern [H-07]
        OfferNFT storage offer = offerNfts[_nft][_tokenId][msg.sender];
        // require(offer.offerer != msg.sender,"Already offer by this
user");
        if(offer.offerer == msg.sender){
        (bool sent, ) = payable(offer.offerer).call{
        value: offer.offerPrice
        }("");
     ...
    }
```

[H-08] Wrong NFT Approval Check Allows Unauthorized NFT Listing

Severity

Impact: Medium

Likelihood: High

Description

In the `MIDNIGHT.sol` contract, NFTs are meant to be approved by the contract owner before they can be listed, as seen in the `ApproveAddressofGhostContract` function on line 288:

```
function ApproveAddressofGhostContract(address ApproveableAddress)
external onlyOwner{
    ApprovedandMinted[ApproveableAddress] = true;
}
```

However, there is a flaw in the require statement on line 345:

```
require(ApprovedandMinted[_nft] = true, "Not the Valid Contract Address");
// @audit-issue this approves any NFTs
```

The single `=` assigns the value `true` to `_nft`, bypassing the intended check and allowing any NFT to be listed, even if it is not approved by the contract owner. This creates a significant security risk, as unapproved NFTs could be listed and potentially sold without meeting the intended approval process.

Recommendations

Modify the require statement to use a double equals (`==`) for comparison:

```
require(ApprovedandMinted[_nft] == true, "Not the Valid Contract
Address");
```

This will ensure that only approved NFTs can be listed on the marketplace.

[H-09] Unauthorized Request for NFT Delivery

Severity

Impact: High

Likelihood: Medium

Description

In the `MIDNIGHT.sol` contract, the `requestDelivery` function (on line 480) allows users to request the delivery of an NFT after the sale has been completed. However, the function does not verify whether the caller is the actual owner of the NFT. This oversight allows unauthorized users to request the delivery of NFTs they do not own, leading to potential exploitation.

Recommendations

Add a check to ensure that only the owner of the NFT can request delivery. Instead of relying on `list.seller`, the token ownership should be verified using the token ID:

```
require(IERC721(listedNft.nft).ownerOf(listedNft.tokenId) == msg.sender,
"Not the NFT owner");
```

This will prevent unauthorized users from initiating the delivery process.

[H-10] Unlimited NFT Delivery Requests Leading to Loss of Funds

Severity

Impact: High

Likelihood: Medium

Description

In the `MIDNIGHT.sol` contract, the `requestDelivery` function (on line 480) allows users to request the delivery of an NFT. However, after the NFT's delivery status is set to "Requested," the contract does not prevent subsequent requests. This issue allows users to repeatedly request delivery for the same NFT, leading to a significant loss of funds as fees are paid for each request.

The lack of a safeguard in this function opens the contract up to exploitation, where users can be charged repeatedly for redundant delivery requests.

Recommendations

Add a check to ensure that the delivery request can only be made once for both auctioned and listed NFTs:

```
require(listedNft.deliveryStatus != DeliveryStatus.REQUESTED, "Delivery
already requested");
require(auction.deliveryStatus != DeliveryStatus.REQUESTED, "Delivery
already requested");
```

This will prevent multiple delivery requests for the same NFT and avoid unnecessary loss of funds for users.

[H-11] Auction NFT Theft via Missing NFT Address Validation

Severity

Impact: High

Likelihood: Medium

Description

The `cancelAuctionWithoutStarting` and `cancelAuctionwhenNoParticipent` functions allow users to cancel their auction and reclaim their NFT. However, there is no validation ensuring that the NFT address (`_nft`) passed to the function matches the NFT associated with the auction ID (`_auctionId`).

This flaw allows a malicious user to hijack auctions and steal NFTs that were auctioned by other users. The attack can be executed by creating a new auction for a token ID that matches an already auctioned token they wish to steal and then calling one of these cancellation functions. Since the `_nft` parameter is not validated, the contract allows the attacker to transfer the token from the contract back to themselves.

Vulnerable Code

```
IERC721 nft = IERC721(_nft);
nft.transferFrom(address(this), msg.sender, auction.tokenId);
```

In both the `cancelAuctionWithoutStarting` and `cancelAuctionwhenNoParticipent` functions, the contract uses the `_nft` parameter passed by the user without verifying that it matches the NFT contract address associated with the auction.

Recommendations

To prevent this vulnerability, ensure that the NFT address being passed matches the NFT address associated with the auction in the contract's storage:

```
require(_nft == auction.nft, "Mismatched NFT address");
```

Add this check in both `cancelAuctionWithoutStarting` and `cancelAuctionwhenNoParticipent` functions to ensure that only the legitimate creator of the auction can cancel and retrieve their NFT.

[M-01] Minimum bid check is not correctly implemented

Severity

Impact: Medium

Likelihood: High

Description

Current implementation of the minimum bid check is wrong. The intended behavior is whenever a new bid is placed then it should be increased by a minimum amount of value from the previous highest bid but currently it is implemented wrongly as can be seen form the below code.

```
function bidPlace(
        address _nft,
        uint256 _auctionId
    ) external nonReentrant isAuction(_auctionId) payable {
        require(
            block.timestamp >= auctionNfts[_auctionId].startTime,
            "auction not start"
        );
        require(
            block.timestamp <= auctionNfts[_auctionId].endTime,
            "auction ended"
        );
         //Audit fix [M-03]
        require(
            msg.value >
                auctionNfts[_auctionId].heighestBid,
            "less than highest bid price"
        );
        require(
            msg.value >=
                    auctionNfts[_auctionId].minBid,
            "less than min bid price"
        );

        AuctionNFT storage auction = auctionNfts[_auctionId];


        if (auction.lastBidder != address(0)) {
            bidderFunds[_auctionId][auction.lastBidder] +=
auction.heighestBid;
        }

        // Set new heighest bid price
        auction.lastBidder = msg.sender;
        auction.heighestBid = msg.value;

        emit PlacedBid(_nft, _auctionId, msg.value, msg.sender);
    }
```

Now the main error in check is as follows

```
 require(
            msg.value >=
                    auctionNfts[_auctionId].minBid,
            "less than min bid price"
        );
```

As can be seen that the msg.value is checked to be greater than min bid here but that is not what the code logic. It should check that the current bid is increased from the previous highest bid by at least `minBid` amount.

Recommendations

Modify the code as follows. Instead of the following checks:

```
 require(
            msg.value >
                auctionNfts[_auctionId].heighestBid,
            "less than highest bid price"
        );
        require(
            msg.value >=
                    auctionNfts[_auctionId].minBid,
            "less than min bid price"
        );
```

replace it with one following check:

```
 require(
            msg.value >
                auctionNfts[_auctionId].heighestBid +
auctionNfts[_auctionId].minBid.
            "lesser than minimum increase in  highest bid price"
        );
```

[M-02] User can avoid using `breedWithRented` by directly calling `breedWithAuto`

Severity

Impact: Medium

Likelihood: Medium

Description

Renter of a NFT can avoid returning the NFT and simultaneously use the functionality of the rented NFT by just calling the breedWithAuto function directly. This causes the renter NFT to be never be returned.

Recommendations

Allow the user to breed with the rented NFT only through the breedWithRented function and not with `breedWithAuto` function.

[M-03] There is no check for breeding starting date for the sire generation

Severity

Impact: Medium

Likelihood: Medium

Description

In breed with auto function it is checked that the block.timestamp >= generationBreedingStartDate[matron.generation] but it is not checked for the sire NFT generation, as sire can be of generation greater than that of the matron so its breeding start date should also be checked.

Recommendations

Implement a similar check for the sire generation as well.

[M-04] `sireAllowedToAddress[_sireId]` is not allowed to call `breed()`

Severity

Impact: Medium

Likelihood: Medium

Description

Sire allowed to address is the address which is allowed to breed on behalf of the owner of that NFT but currently there is no code implementation which allows the sire allowed address to perform breeding with other NFTs thus rendering this functionality redundant.

Recommendations

In `breedWithAuto` function allow the `sireAllowedToAddress[_sireId]` to breed with other matron NFTs. One way can be like as follows:

```
require((ownerOf(_matronId) == msg.sender &&
        rentals[_sireId].renter == msg.sender) ||
        (ownerOf(_matronId) == msg.sender &&
        ownerOf(_sireId) == msg.sender) || (ownerOf(_matronId) ==
msg.sender && sireAllowedToAddress[_sireId] == msg.sender)  , "Not owner
nor renter");
```

[L-01] `getAllAuctions` function uses incorrect `auctionNfts` mapping

Severity

Impact: Low

Likelihood: Low

Description

Following is `getAllAuctions` function:

```
 function getAllAuctions(uint256 _startId, uint256 _endId) external view
returns (AuctionNFT[] memory _allAuctions) {
        require(_startId <= _endId && _endId < auctionCount, "invalid
range");
        _allAuctions = new AuctionNFT[](_endId - _startId + 1);
        for (uint256 i = _startId; i <= _endId; i += 1) {
            _allAuctions[i - _startId] = _allAuctions[i];
        }
    }
```

The following line is incorrect:

```
_allAuctions[i - _startId] = _allAuctions[i];
```

Instead of accessing `auctionNfts` mapping it accesses `allAuctions` mapping.

Recommendations

Modify the code as follows:

```
function getAllAuctions(uint256 _startId, uint256 _endId) external view
returns (AuctionNFT[] memory _allAuctions) {
        require(_startId <= _endId && _endId < auctionCount, "invalid
range");
        _allAuctions = new AuctionNFT[](_endId - _startId + 1);
        for (uint256 i = _startId; i <= _endId; i += 1) {
      ---      _allAuctions[i - _startId] = _allAuctions[i];
      +++  _allAuctions[i - _startId] = auctionNfts[i];
```

```
        }
    }
```

[L-02] Predictable Randomness in Winner Selection

Description

The `randomWinners` function in `GHOSTRAFFLE.sol` uses a weak source of randomness to select raffle winners. The function relies on `block.timestamp`, `block.difficulty`, and other on-chain values to generate random indices for winner selection. However, these values are predictable and can be manipulated by malicious actors, such as miners or participants, to alter the outcome of the raffle.

This weak source of randomness increases the risk of the raffle results being manipulated, leading to unfair and biased selection of winners.

Vulnerable Code

```
uint256 winnerIndex = uint256(keccak256(abi.encodePacked(block.timestamp,
block.difficulty, msg.sender, i))) % ticketOwners.length;
```

The use of `block.timestamp` and `block.difficulty` in the `keccak256` hash calculation is a well-known anti-pattern for generating random numbers, as these values can be predicted or influenced by miners.

Recommendations

It is highly recommended to use an external verifiable source of randomness such as Chainlink VRF (Verifiable Random Function) or another trusted oracle service. These oracles provide unpredictable and tamper-resistant randomness, ensuring that the raffle outcome cannot be manipulated.

An example of integrating Chainlink VRF for randomness generation:

1. Request random number via Chainlink VRF.
2. Use the random number to select winners from the list of `ticketOwners`.
3. Ensure that the random number is used only after it is returned by the oracle to prevent any manipulation.

For more information, refer to Chainlink VRF's documentation on how to securely implement randomness: Chainlink VRF Documentation.