e

CD SECURITY

AUDIT REPORT

Midnight

September 2024

Prepared by
yotov721
Arnie

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* - 8fbbb7fe27c704e5551322a8f3b3c9b77232df2d**

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: 13 issues
- Medium: 9 issues
- Low: 12 issues

Findings Summary

ID	Title	Severity	Status
[C-01]	Anyone can mint the entire NFT collection and set genes	Critical	Fixed
[C-02]	Malicious user can bid the minimum and always win	Critical	Fixed
[C-03]	Attacker could force win auction by reverting on higher bids	Critical	Fixed
[C-04]	Direct theft of NFT through `cancelListedNFT`	Critical	Fixed
[H-01]	Participating in raffle with multiple tickets does not work	High	Fixed
[H-02]	The total supply of NFTs might not be fully minted	High	Fixed
[H-03]	NFT ids do not correspond to their position in the nfts array	High	Fixed
[H-04]	NFT marked as sold on initial sale can lead to losing funds	High	Fixed
[H-05]	Users can place bets on canceled auctions	High	Fixed
[H-06]	`acceptOfferNFT` can be frontrun to scam the seller	High	Fixed
[H-07]	Making more than 1 offer on an item will cause loss of eth	High	Fixed
[H-08]	Direct theft of initial sale NFT	High	Fixed
[H-09]	`buyNFT` does not check if a listing id was canceled	High	Fixed
[M-01]	A renter can DoS owner from transferring his NFT	Medium	Fixed
[M-02]	NFT breeding cooldown is calculated wrong	Medium	Fixed
[M-03]	A user can make a higher bet without increasing the bet price	Medium	Fixed
[M-04]	No way to remove token from being payable	Medium	Fixed
[M-05]	Users can list NFT from other collections if they own ghost NFT	Medium	Fixed

ID	Title	Severity	Status
[M-06]	Winning index can be selected more than once in the raffle	Medium	Fixed
[M-07]	NFT can be stuck in the contract	Medium	Fixed
[M-08]	`resultAuction` doesn't correctly validate the `_nft` param	Medium	Fixed
[M-09]	Flawed modulo use of the amount of `tickOwners.length`	Medium	Fixed
[L-01]	Use call instead of transfer to send ETH	Low	Fixed
[L-02]	A user can breed two rented NFTs	Low	Fixed
[L-03]	Users can breed NFTs before start time is set	Low	Fixed
[L-04]	Unused logic in `GHOSTBREEDING` contract	Low	Fixed
[L-05]	Genes are NOT selected at random	Low	Fixed
[L-06]	`isListedNFT` does not correctly verify listed NFTs	Low	Fixed
[L-07]	Condition `isCheckInitialSale` always evaluates to true	Low	Fixed
[L-08]	Wrong age returned if NFT does NOT exist	Low	Fixed
[L-09]	When auction ends winner is set to creator, not bidder	Low	Fixed
[L-10]	Extra pay amount NOT refunded when buying NFT	Low	Fixed
[L-11]	Pay token is never used to pay for listed NFTs	Low	Fixed
[L-12]	Off by one issue at `getValidAuctions`	Low	Fixed

Detailed Findings

[C-01] Anyone can mint the entire NFT collection and set genes

Severity

Impact: High

Likelihood: High

Description

The protocol's main concept is around NFTs. One of the contracts is a classic NFT contract with additional features like generations, breeding, genes, and lending the NFTs. The problem however is that the `mint` function lacks access control and anyone can mint the entire supply of generation 0 to himself, right after deployment. The malicious user could even pass desired genes.

The `mint` function calls `_mintSingleNFT()` which does the actual minting

Recommendations

Let only the owner be able to mint. Add the `onlyOwner` modifier to the `NFT_ERC721::mint()` function.

[C-02] Malicious user can bid the minimum and always win

Severity

Impact: High

Likelihood: High

Description

```
    function bidPlace(
        address _nft,
        uint256 _auctionId
    ) external payable nonReentrant isAuction(_auctionId) {
        require(
            block.timestamp >= auctionNfts[_auctionId].startTime,
            "auction not start"
        );
        require(
            block.timestamp <= auctionNfts[_auctionId].endTime,
            "auction ended"
        );

        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];
        // IERC20 payToken = IERC20(auction.payToken);
        // payable().transferFrom(msg.sender, address(this), _bidPrice);

        if (auction.lastBidder != address(0)) {
            address lastBidder = auction.lastBidder;
            uint256 lastBidPrice = auction.heighestBid;

            // Transfer back to last bidder
            payable(lastBidder).transfer(lastBidPrice);
        }
```

the bidPlace function allows a user to place a bid on an auction, if the bid is higher than the previous bidder, the previous bidder will receive his ETH back via a transfer.

The problem occurs when a malicious user uses a contract to place a bid. The malicious user adds a fallback function that reverts when receiving ether, therefore whenever he bids, no one may bid after him because the call will always revert.

This allows the attacker to bid the minimum amount and never be outbid and thus cheat the auction creator out of his deserved funds.

Additionally, since the nft will be transferred to him after the auction, he can actually receive the nft with no problem.

Recommendations

Implement a pull mechanism, where the bidders can withdraw their bids in another function.

[C-03] Attacker could force win auction by reverting on higher bids

Severity

Impact: High

Likelihood: High

Description

The protocol lets users create auctions and other users bet on them for a limited time. When a user creates a higher bid than the last, the last user's funds are returned. The problem is the way they are returned - the funds are directly sent back to the using `transfer`. The problem with this is that a malicious user could bid through a contract and set its `receive` and `callback` functions to revert when they receive ETH. That way the whole transaction would revert and it would NOT be able to outbid him. This gives an unfair advantage and is clearly wrong.

```
        if (auction.lastBidder != address(0)) {
            address lastBidder = auction.lastBidder;
            uint256 lastBidPrice = auction.heighestBid;

            // Transfer back to last bidder
            payable(lastBidder).transfer(lastBidPrice);
        }
```

Note that this is also present, but with less severity, in the `acceptOfferNFT`, `buyNFT` and `resultAuction` functions

Recommendations

The most robust solution here would be to use pull over push - meaning to save the amount owned to a user and create a function that lets his pull it out of the contract. A mapping of address user => uint256 amount would do a perfect job.

[C-04] Direct theft of NFT through `cancelListedNFT`

Severity

Impact: High

Likelihood: High

Description

A malicious user can directly steal any NFT in the marketplace because the `_nft` param is not correctly validated by the function. Let us take a look at the snippet...

```
    function cancelListedNFT(
        address _nft,
        uint256 _listingId
    ) external nonReentrant isListedNFT(_listingId) {
        ListNFT memory listedNFT = listNfts[_listingId];
        require(listedNFT.seller == msg.sender, "not listed owner");
        IERC721(_nft).transferFrom(
            address(this),
            msg.sender,
            listedNFT.tokenId
        );
        // delete listNfts[_listingId];
        listNfts[_listingId].status = Status.CANCELLED;
    }
```

the function allows a user to cancel a listing of their nft, the user must input the nft address and the listing id. The problem occurs because the code never validates that `_nft` address is the correct address of the nft they have listed.

the attacker can list a low-value nft of any token id, and then when they want to steal another NFT from a different collection but with the same token id, the attacker can simply call `cancelListedNFT` with a different `_nft` address that is not the address of his NFT, the logic will then transfer him the NFT of another address but same token id that was listed by another user and hence held by the contract.

Because the logic does not check if the status of the listing id is already canceled, the attacker may drain every NFT of every collection in the contract with the same token id, he may then withdraw his NFT.

Essentially an attacker can drain every NFT in the protocol if he just deposits the same token id and then inputs the different `_nft` address.

Recommendations

Validate `_nft` param is the same as the NFT address in the listing to ensure a user cannot steal another user's NFT.

[H-01] Participating in raffle with multiple tickets does not work

Severity

Impact: High

Likelihood: Medium

Description

The protocol has a raffle where users can buy tickets with ETH in exchange for the chance to win an NFT from the GHOST collection. There are 1000 tickets in total and 10 winners. A user may buy multiple tickets. Hence the same address/user could win more than one NFT. The 10 winners are chosen at random using Chainlink VRF. After that, they can redeem their NFT using the `GhostNFTMarketplace::initialSale` function.

If we take a look at the function it uses merkle tree verification, to verify the buyer is a winner, where the leaf is the hash of the winning address.

```
    function initialSale(
        uint256 _tokenId,
        address _ownerAddress,
        bytes32[] calldata proof
    ) external payable nonReentrant {
        require(msg.value == 1 ether, "Invalid Price");
        require(_tokenId < 10, "Invalid Token Id");

        require(claimed[msg.sender] == false, "already claimed");
        claimed[msg.sender] = true;
        require(
            MerkleProof.verify(proof, merkleRoot, toBytes32(msg.sender))
== true,
            "Invalid user"
        );

        _processInitialSale(msg.value);
        IERC721(GhostNFTAddress).safeTransferFrom(
...
        );
    }
```

We can also see that the `claimed` mapping is updated and a winner can NOT claim more than once. Having all this in mind the problem is that a user can win more than one NFT from the raffle, but can claim at most one. If a user has two or more winning tickets he could claim only 1 NFT.

Recommendations

In order to fix this it is best to hash the winning address and the NFT id into the leaf. To track who has withdrawn his reward a mapping of address => uint256 nftId => bool can be used.

[H-02] The total supply of NFTs might not be fully minted

Severity

Impact: High

Likelihood: High

Description

The protocol implements a classic NFT contract with the addition of generations. The first generation is the 0th generation and it is the only one that can be minted. All other generations are minted only via breeding of previous generations.

In the current state, the total supply for the 0th generation is 150. However, there is a scenario in which the total amount may NOT be minted and it may not be possible for it to be minted. If we take a look at the `mint` function there are two branches that can be used to mint depending on the `isBatch` flag.

```
if (isBatch) {
    // Batch minting logic
    require( //                                          100 / 10
        mintedBatches < BATCH_SUPPLY / BATCH_SIZE,
        "All batches have been minted"
    );
    require(quantity == BATCH_SIZE, "Batch size must be 10");
    require(
        totalMinted + quantity <= BATCH_SUPPLY, // 100
        "Exceeds batch supply limit"
    );
    for (uint i = 0; i < quantity; i++) {
        _mintSingleNFT(msg.sender, tokenUri[i], genes[i]);
    }
    mintedBatches += 1;
} else {
    // Release-based minting logic
    require(totalReleases <= 5, "Exceeds total releases");
    require(
        totalMinted + quantity <= TOTAL_SUPPLY,
        "Exceeds total supply"
```

```
        );
        require(quantity > 0 && quantity <= 10, "Invalid release size");
        for (uint i = 0; i < quantity; i++) {
            _mintSingleNFT(msg.sender, tokenUri[i], genes[i]);
        }
        totalReleases += 1;
        releaseSizes[totalReleases] = quantity;
    }
```

If the `isBatch` flag is set to `true` 10 NFTs are minted. There are 10 batches in total - so 100 NFTs can be minted via batch. The other branch of minting is release-based where there can be a maximum of 6 releases for 50 NFTs left. Here however a user may mint only a single NFT. If the 50 NFTs that are NOT minted from these 6 releases they are lost forever. This is enforced by the `require(totalReleases <= 5, "Exceeds total releases");` statement, and the `totalReleases` variable is incremented at the end of the function.

Another way to NOT mint all NFTs is again if less than 10 NFTs are minted by the release-based minting logic before the batch minting logic. This is caused because the batch minting logic compares the totalMinted amount plus the current quantity against the batch supply, NOT the total supply.

Recommendations

Remove the total releases max 5 requirement, as the total supply is kept in check by the `require(totalMinted + quantity <= TOTAL_SUPPLY, "Exceeds total supply");` statement.
When batch minting, compare the `totalMinted` + `quantity` against the total supply.
The maxim minted by batches is already enforced by the `mintedBatches < BATCH_SUPPLY / BATCH_SIZE` require statement.

```
    if (isBatch) {
                // Batch minting logic
                require(mintedBatches < BATCH_SUPPLY / BATCH_SIZE, "All
    batches have been minted");
                require(quantity == BATCH_SIZE, "Batch size must be 10");
    -           require(totalMinted + quantity <= BATCH_SUPPLY, "Exceeds batch
    supply limit");
    +           require(totalMinted + quantity <= TOTAL_SUPPLY, "Exceeds batch
    supply limit");
                for (uint i = 0; i < quantity; i++) {
                    _mintSingleNFT(msg.sender,tokenUri[i],genes[i]);
                }
                mintedBatches += 1;
            } else {
                // Release-based minting logic
    -           require(totalReleases <= 5, "Exceeds total releases");
                require(totalMinted + quantity <= TOTAL_SUPPLY, "Exceeds total
    supply");
                require(quantity > 0 && quantity <= 10, "Invalid release
    size");
                for (uint i = 0; i < quantity; i++) {
                    _mintSingleNFT(msg.sender,tokenUri[i],genes[i]);
```

```
            }
            totalReleases += 1;
            releaseSizes[totalReleases] = quantity;
        }
```

[H-03] NFT ids do not correspond to their position in the nfts array

Severity

Impact: High

Likelihood: High

Description

When a new NFT is minted it is assigned an id which is pushed in the `nfts` array. Afterward, NFTs are accessed in that array by id as if the id corresponds to the NFT index in the array. There are two problems with this logic. First, the array index starts at `0` and the first NFT id is `1`, because the counter is first incremented and assigned as id to the newly minted NFT after that. Hence it would equal `1`.
The second issue is with the generation after the initial one. Basically, the protocol mints only the 0th generation which is 150 NFTs and every single one after is minted only by breeding. When a new NFT is born it is assigned an `id` from a new state variable counter called `length`, which is initially equal to `150`. The problem here is that a NFT can be born before all NFTs from the 0th generation are minted. Hence it would be added to the array and be at an index below `150` but its id would be above `150`.
This discrepancy between NFT id in the contract and in the array index completely breaks all the protocol logic.

Recommendations

The most robust solution would be to use a mapping like `mapping(uint256 nftId => NFT)` to store and access the NFT data.

[H-04] NFT marked as sold on initial sale can lead to losing funds

Severity

Impact: High

Likelihood: High

Description

The protocol implements a classic NFT contract with the addition of a marketplace where users could sell and auction off their NFTs. When an NFT is bought (or auction settled) there is a check that verifies if this is

a first sale.

```
if (!nftSold[_nft][_tokenId]) {
  // If it's an initial sale
  _processInitialSale(totalPrice);
  nftSold[_nft][_tokenId] = true;
} else {
  // If it's a resale, process royalty
  totalPrice = _processResale(totalPrice);
  payable(auction.creator).transfer(totalPrice);
}
```

Basically, if it is an initial sale, from the protocol, the payment is transferred to the protocol wallets. If it is a re-sale the funds are transferred to the seller's wallet.

There are also 10 users who by participating in a raffle have won the chance to buy one of the first 10 GHOST NFTs. This is implemented in the `GhostNFTMarketplace::initialSale` function. The problem is that on the initial sale the `nftSold` is NOT updated and when a user sells or auctions off the NFT through the `GhostNFTMarketplace` the payment would go to the protocol wallets instead of the seller.

Recommendations

In `GhostNFTMarketplace::initialSale` update the `nftSold` mapping to `true`.

```
+    nftSold[GhostNFTAddress][_tokenId] = true;
```

[H-05] Users can place bets on canceled auctions

Severity

Impact: High

Likelihood: Medium

Description

The protocol lets users create auctions and sell off their NFTs. Sellers can also cancel the auction but only if it is before the auction has started. This is achieved via the `cancelAuction` function and only the auction status is updated to `CANCELLED` and the NFT is transferred back to the owner. In the `bidPlace` and `resultAuction` functions there is no check whether the auction was canceled. That lets users freely bet on canceled auctions and when the time comes to close the auction the transaction will revert because the NFT was sent back to the owner when canceling. The last bidder's funds would get stuck in the contract.

Recommendations

Delete the auction data in `cancelAuction`.

```
        nft.transferFrom(address(this), msg.sender, auction.tokenId);
        auctionNfts[_auctionId].status = Status.CANCELLED;
+       delete auctionNfts[_auctionId];
    }
```

[H-06] `acceptOfferNFT` can be frontrun to scam the seller

Severity

Impact: High

Likelihood: Medium

Description

```
    function acceptOfferNFT(
        address _nft,
        uint256 _tokenId,
        address _offerer,
        uint256 _listingId
    )
        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[offer.listingId];
        require(!list.sold, "Already sold");
        require(!offer.accepted, "Offer already accepted");

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

        if (!nftSold[_nft][_tokenId]) {
            _processInitialSale(offerPrice);
            nftSold[_nft][_tokenId] = true;
        } else {
            offerPrice = _processResale(offerPrice);
            payable(list.seller).transfer(offerPrice);
        }
```

The acceptOfferNFT function allows a seller to accept an offer from a user to sell their nft.

The problem occurs because the code allows the offerer to front run this tx when he sees it in mempool. Let us say the owner sees an offer for 1 eth and then calls the function to accept the offer. The attacker can frontrun the tx with 2 transactions, one where he cancels the current offer, and the second tx he submits a new offer for a very low amount, 1 wei.

The victim user's tx will now pass and will accept the 1 wei offer even when he was trying to accept the 1 ether offer.

The user was essentially robbed of his nft.

This is possible because the OfferNFT mapping is updated and can be frontrun to do this attack.

Recommendations

Add mechanisms such as offer ids to ensure the frontrun attack cannot happen. For example, offer id 1 was accepted but offer id 2 was not. This means that if offer id 1 is canceled, the user is not forced to sell to offer id 2. Furthermore, the protocol may also add a param named minimumAmount which validates the user is receiving at least the minimumAmount desired.

[H-07] Making more than 1 offer on an item will cause loss of eth

Severity

Impact: High

Likelihood: Medium

Description

```
    function offerNFT(
        address _nft,
        uint256 _listId,
        address _payToken
    ) external payable isListedNFT(_listId) nonReentrant {
        require(msg.value > 0, "price can not 0");
        // require(_tokenId > 10, "NFT is already sold through Raffle");

        ListNFT memory nft = listNfts[_listId];

        offerNfts[_nft][nft.tokenId][msg.sender] = OfferNFT({
            nft: nft.nft,
            tokenId: nft.tokenId,
            offerer: msg.sender,
            payToken: _payToken,
            offerPrice: msg.value,
            accepted: false,
```

```
            listingId: _listId
        });
```

the offerNFT function allows a user to make an offer on an nft. The function is payable and the user must send eth via msg.value in order to make the offer. The problem occurs because the function does not correctly handle the case where the user plans to make more than 1 offer on the nft as is allowed on all marketplaces.

for example let us say the user puts a 1 eth offer, he sends over 1 eth in msg value and the mapping is updated `offerNfts[_nft][nft.tokenId][msg.sender] = OfferNFT({`

the user wants to place another offer because his previous offer was to low, this time he sends 2 eth, the mapping is overriden with the new data. `offerNfts[_nft][nft.tokenId][msg.sender] = OfferNFT({`

the previous data is lost so there is no way for the user to withdraw his previous deposit of 1 eth only the 2 eth.

```
    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];
        payable(offer.offerer).transfer(offer.offerPrice);
        emit CanceledOfferredNFT(
            offer.nft,
            offer.tokenId,
            offer.payToken,
            offer.offerPrice,
            msg.sender
        );
```

above we can see the function will transfer the funds back according to the current mapping, the overridden data will not be able to retrieve the user 1 eth and is now lost forever.

Recommendations

Either implement logic that will block making an offer if an offer on the item is currently active or add logic to handle multiple offers.

[H-08] Direct theft of initial sale NFT

Severity

Impact: High

Likelihood: High

Description

```
    function initialSale(
        uint256 _tokenId,
        address _ownerAddress,
        bytes32[] calldata proof
    ) external payable nonReentrant {
        require(msg.value == 1 ether, "Invalid Price");
        require(_tokenId < 10, "Invalid Token Id");

        require(claimed[msg.sender] == false, "already claimed");
        claimed[msg.sender] = true;
        require(
            MerkleProof.verify(proof, merkleRoot, toBytes32(msg.sender))
==
                true,
            "Invalid user"
        );

        _processInitialSale(msg.value);

        IERC721(GhostNFTAddress).safeTransferFrom(
            _ownerAddress,
            msg.sender,
            _tokenId
        );
    }
```

The `intitialSale` function is the sale of the first 10 token ID to whitelisted individuals, the problem occurs because a user can claim a token id that has already been claimed.

The following can be done by inputting the `_ownerAddress` as the victim/ current owner of the desired token id, then the merkle proof simply verifies that the msg.sender is valid.

The final part of the logic will then transfer the inputted token id from the `_ownerAddress` to the msg.sender even if said token id is already claimed.

This is possible if the user has given approval to the contract which is expected because the user will need to give approval to list his NFT.

therefore a user can steal the token id 1 which may be more valuable from an innocent victim.

Recommendations

Validate that the token id has already been bought in the initial sale in order to stop the theft of NFT.

[H-09] buyNFT does not check if a listing id was canceled

Severity

Impact: High

Likelihood: High

Description

The `buyNFT` function in `MIDNIGHT.sol` In 445:

```
    function buyNFT(
        address _nft,
        uint256 _listId,
        address _payToken
    ) external payable nonReentrant isListedNFT(_listId) {
        ListNFT storage listedNft = listNfts[_listId];
        // require(_tokenId > 10, "NFT is already sold through Raffle");
        require(
            _payToken != address(0) && _payToken == listedNft.payToken,
            "Invalid pay token"
        );
        require(!listedNft.sold, "NFT already sold");
        require(msg.value >= listedNft.price, "Invalid price");

        listedNft.sold = true;
        listedNft.status = Status.COMPLETED;
        uint256 totalPrice = msg.value;

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

the function above allows a user to buy an nft that is listed, the problem occurs because the function never does validate that the nft lisiting has been canceled or not.

For example let us say a user has listed an nft for 1 eth, the lister canceled the listing and relisted the nft for 10 eth because his first listing was either done a long time ago or was a mistake.

Althought the lister has updated the price and canceled the previous listing id. A malicious user can still buyNFT with the canceled listing id in order to buy the nft for the cheaper price of 1 eth intead of 10 eth even if the lister has canceled the listing.

Recommendations

`buyNFT` function should validate if the the listing was canceled.

[M-01] A renter can DoS owner from transferring his NFT

Severity

Impact: Medium

Likelihood: Medium

Description

The protocols let users rent their NFTs for a price set by them. Renting the NFT lets the renter use it to breed with his own NFT and give birth to a new one. Until the NFT is rented the owner can NOT transfer it. Once the NFT is bred or the rented just decides - it can be returned. The problem is in the `rentNFT` function because it lets the renter input the rent time in days. Hence a malicious user can input a large number of days and NOT breed or return the NFT from rent, intentionally griefing the owner from transferring it. This also locks the NFT from being bred with other NFTs.

Recommendations

Add a maximum rent time that can NOT be exceeded - 1 or 2 days for example.

[M-02] NFT breeding cooldown is calculated wrong

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol lets users breed their NFTs. Two NFTs can only be bred together if their `cooldownEndBlock` is less than `block.number` as seen below:

```
function _isReadyToBreed(NFT storage nft) internal view returns (bool) {
    return (nft.siringWithId == 0) && (nft.cooldownEndBlock <=
block.number);
}
```

Additionally, an NFT only gives birth if the same cooldown is less than `block.number` - enforced by the `_isReadyToGiveBirth` function.

The problem is how this cooldown is calculated. Taking a look at the only function that it is updated in we can see that it actually calculates the block that would end the cooldown.

```
function _triggerCooldown(NFT storage _nft) internal { // ok
    _nft.cooldownEndBlock = uint64(
        (cooldowns[_nft.generation] / secondsPerBlock) + block.number
    );
}
```

The problem here, however, is that `secondsPerBlock` is set `15`, while the average block time on the Pume Network is ~0.5 seconds (This can be verified on the plume testnet's official website: https://plume-testnet.explorer.caldera.xyz/). Having in mind the `secondsPerBlock` is set to 15 which is around 30 times more than the network block time this completely breaks the NFT cooldown logic. Also, it is bad in general to use `block.number` for time tracking operations as it is certainly NOT constant. Block time bay be 0.4 seconds causing the cooldowns to pass faster or 0.6 seconds (if there is less activity) causing cooldown times to pass slower.

Recommendations

The most robust solution would be to use `block.timestamp` instead of blocks to track the NFT cooldown.

[M-03] A user can make a higher bet without increasing the bet price

Severity

Impact: Medium

Likelihood: High

Description

The protocol lets users auction off their NFTs. The owner can set min bet amount. Users can put bets and at the end of the auction the highest bidder gets the NFT. The problem is that a user may bet the same amount as the current highest bet and be the highest bidder. This is possible because of the way the highestBid is compared:

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

Recommendations

Change `GhostNFTMarketplace::bidPlace` to so it only updates the highest bidder if the new bid is higher, not equal.

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

[M-04] No way to remove token from being payable

Severity

Impact: High

Likelihood: Low

Description

The protocol lets users auction off their NFTs. Users can choose the auction currency - native or some ERC20/payable token. There is a payable token whitelist that defines the tokens that are allowed to be used in the auctions. Whitelisted pay tokens can only be added by the admin. However, there is no functionality that removes a payable token flow being allowed. This is a must since the token may get hacked, drained, or rugged. A good example of this is the TERRA/LUNA stablecoin collapse.

Recommendations

Implement a function that lets the admin delist a pay token.

[M-05] Users can list NFT from other collections if they own ghost NFT

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol lets users list their NFTs for sale

```
    function listNft(
        address _nft,
        uint256 _tokenId,
        address _payToken,
        uint256 _price
    )
        external
        isPayableToken(_payToken)
        onlyMintedByGhostNFTContract(_tokenId)
        nonReentrant
    {
        IERC721 nft = IERC721(_nft);
...
        nft.transferFrom(msg.sender, address(this), _tokenId);
...
```

As per the comment, there would be a few NFT collections that could be listed for sale, the first one being the Ghost one. The problem with the current listing function is that a user can list an NFT from any other collection for sale as long as he has an NFT from the GHOST collection with the same id and has approved it to the marketplace contract.

The reason behind this is that the user inputs the NFT collection address and NFT id. However, in the `onlyMintedByGhostNFTContract` modifier it is only checked if the user has allowed the marketplace contract to operate with the NFT in mind. That way a user can just own a GHOST NFT with some id and have a set allowance to the marketplace contract, but pass a different collection NFT address and `_tokenId`. The NFT from the fake collection would be transferred to the marketplace contract and up for sale.

Recommendations

Add a `mapping(address => bool)` of NFT contract addresses that are whitelisted and check if the user passed NFT contract address is whitelisted. That way an admin can easily add or remove whitelisted NFT contracts.

[M-06] Winning index can be selected more than once in the raffle

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol lets users join a raffle in exchange for the chance to win an NFT. To enter the raffle a user needs to pay the ticket price after which he is minted an NFT representing his position. A user may have

multiple entries with the same address. The contract uses Chainlink VRF to select the winners.

```
for (uint256 i = 0; i < numWinners; i++) {
    // Find random winners and store them in array;
    uint256 winnerIndex = randomResults[i] % ticketOwners.length;
    winners.push(ticketOwners[winnerIndex]);
}
```

The problem is when they are picked. If two or more of the random results end on the same 3 digits that would pick the same winning index more than once. This would cause a user to get 2 NFTs for one ticket.

Recommendations

Remove the winning index from the array by making it equal to the last element from the array and the call the `pop()` method to remove the last.

```
+     ticketOwners[winnerIndex] = ticketOwners[ticketOwners.length - 1];
+     ticketOwners.pop();
```

[M-07] NFT can be stuck in the contract

Severity

Impact: High

Likelihood: Medium

Description

The protocol lets users create auctions where they can sell their NFTs. Sellers can set minimum bid price auction start and end times. Sellers can also cancel their auctions, but only if it is before the auction start time. However, if no one participates in the auction and it has ended the last/highest bidder would be the 0 address. There is also no way for the owner to retrieve his NFT back, leaving it stuck in the contract.

Recommendations

Let auction creators cancel an auction if the `lastBidder` == `address(0)` and `block.timestamp` > `endTime`.

[M-08] `resultAuction` doesn't correctly validate the `_nft` param

Severity

Impact: Medium

Likelihood: Medium

Description

```
    function resultAuction(
        address _nft,
        uint256 _auctionId,
        uint256 _tokenId
    ) external nonReentrant {
        AuctionNFT storage auction = auctionNfts[_auctionId];

        require(!auction.success, "already resulted");
        require(
            msg.sender == owner() ||
                msg.sender == auction.creator ||
                msg.sender == auction.lastBidder,
            "not creator, winner, or owner"
        );
        require(block.timestamp > auction.endTime, "auction not ended");

        IERC721 nft = IERC721(auction.nft);

        auction.success = true;
        auction.winner = auction.creator;
        auction.status = Status.COMPLETED;

        uint256 heighestBid = auction.heighestBid;
        uint256 totalPrice = heighestBid;

        if (!nftSold[_nft][_tokenId]) {
            // If it's an initial sale
            _processInitialSale(totalPrice);
            nftSold[_nft][_tokenId] = true;
        } else {
            // If it's a resale, process royalty
            totalPrice = _processResale(totalPrice);
            payable(auction.creator).transfer(totalPrice);
        }
```

the resultAuction function allows a user to buy an nft that was listed for sale, the user will input `_nft` param

The problem occurs because the code never validates that the `_nft` param is, in fact, the correct address according to the auctionNFT. `_nft` param value is only used to determine if the sale is the initial sale or a re-sale.

A user can take advantage of this in order to either only pay the initial sale fee instead of the re-sale fee or vice versa.

Recommendations

Correctly validate the `_nft` param is equal to the auctionNFT nft param.

[M-09] Flawed modulo use of the amount of `tickOwners.length`

Severity

Impact: Medium

Likelihood: Medium

Description

```
    function randomWinners() external onlyOwner {
        require(randomResultRequested == true, "Ghost: Cannot choose
winners without having collected random numbers.");
        require(winners.length == 0, "Ghost: Winners already selected.");
        // require(_tokenIdCounter == MAX_TICKETS, "Ghost: All tickets
must be sold.");
        require(block.timestamp >= raffleEndTime, "Ghost: Raffle time has
not ended.");

        // Store the addresses of the random winners
        for (uint256 i = 0; i < numWinners; i++) {

        // Find random winners and store them in array
            uint256 winnerIndex = randomResults[i] % ticketOwners.length;
            winners.push(ticketOwners[winnerIndex]);
        }

        emit RaffleWon(winners);
    }
```

In the randomWinners function the randomness first gets the randomResults that was fulfilled by chainlink. Then we take the modulo of the value with the `ticketOwners.length`. This will result in a not-so-good source of randomness and will also result in unfair chances depending on the index value of a ticket owner.

For instance, let us assume the `ticketOwners.length` is a high number like 897 and the randomness request is a larger number 24234234234. Because the larger numbers you go, the chance that modulo will return 0 decreases, it creates an unfair scenario for the ticket owner at the lower index values.

Recommendations

Do not allow randomness from user-controlled values such as `ticketOwners.length`.

[L-01] Use call instead of transfer to send ETH

The protocol uses `transfer()` to send ETH on several occasions.

The problem with using `transfer()` to send ETH is that it forwards only 2300 gas and if the receiver is a contract that has a `receive()` or `fallback()` function the 2300 gas could now be enough. Such contracts may be multi-sig wallets.

Use `call()` to send ETH and make sure to add the `nonReentrant` modifier as `call` opens the possibility of reentrancy.

[L-02] A user can breed two rented NFTs

Description

The protocol lets users mint NFTs with the addition of NFT generations starting from the 0th generation. New generations after the 0th can be minted by breeding the previous generation. In order to breed a user must have a matron NFT and rent a sire NFT from another user OR own both. Note that the "matron" and "sire" represent just an nft and can be interchanged, but one needs to be a matron and one sire when breeding.

In the `breedWith` function (which is called at the end of `breedWithAuto`) there are two statements that require the user to own or have rented the matron NFT and the same conditions for the sire. However, in the `breedWithAuto` function there is only a statement that requires the user to own the matron OR have rented the siren.

```
require(ownerOf(_matronId) == msg.sender ||
  rentals[_sireId].renter == msg.sender, "Not the owner of the matron");
```

This opens the possibility of a 3rd party renting both NFTs and breeding them. This should not be possible.

Recommendations

Remove the require check from `breedWithAuto`:

```
-require(ownerOf(_matronId) == msg.sender || rentals[_sireId].renter ==
msg.sender,
-    "Not the owner of the matron"
-);
```

Remove the require statement in `breedWith` and add the following to `breedWithAuto`:

```
require((ownerOf(_sireId) == msg.sender &&
  rentals[_matronId].renter == msg.sender) ||
  (ownerOf(_matronId) == msg.sender &&
    ownerOf(_sireId) == msg.sender), "Not owner nor renter"); // Either
own the matron and rent the sire OR own both
```

[L-03] Users can breed NFTs before start time is set

Description

The protocol lets users breed their NFTs. Each NFT has a generation. Only the 0th generation can be minted and all the next ones are created by breeding. Each generation has a date before which the breeding can not begin stored in the `generationBreedingStartDate` mapping. The values in the mapping can only be set by the admin. The problem is that users can breed before the values in the mapping are set since the default value is zero and the bellow condition is always `true`:

```
require(block.timestamp >=
  generationBreedingStartDate[
    matron.generation
  ], "Breeding has not started for this generation");
```

Recommendations

Set initial values to `generationBreedingStartDate` in the constructor or directly in the contract. They can be far in the future or even `type(uint256).max` in order to make sure no one breeds until the admin sets the desired values.

[L-04] Unused logic in `GHOSTBREEDING` contract

Description

The protocol lets users own and breed their NFTs. When two NFTs are bred a new NFT is minted to the Dame NFT ("mother" NFT) owner. If a user does NOT have two NFTs to breed he can rent an NFT from another user in exchange for payment amount set by the renter. There are a few checks when a user breeds NFTs like does he own one and rent another, that they are different NFTs - not the same. There is a function that is called `_isSiringPermitted` which basically requires the renter to have set an additional flag that lets the user breed with his NFT. This function however is only called the `external view` function `canBreedWith` and NOT in any of the breeding logic.

Recommendations

Remove the "is allowed to sire" logic in general as the user pays the renter to breed with his NFT. No need to have his permission.

[L-05] Genes are NOT selected at random

Description

The protocol lets users own and breed NFT. When two NFTs are bred their genes are mixed to get the child's genes. The problem is that the initial generation's genes are NOT generated at random, but passed as a variable to the `mint` function. This could lead to some users getting NFT with more rare genes than other users. This would make the game unfair for some and more favorable for users with NFTs that have rarer genes.

Recommendations

Use Chainlink VRF to generate genes for the NFTS.

[L-06] `isListedNFT` does not correctly verify listed NFTs

Description

```
    modifier isListedNFT(uint256 _listingId) {
        ListNFT memory listedNFT = listNfts[_listingId];
        require(
            listedNFT.seller != address(0) && !listedNFT.sold,
            "not listed"
        );
        _;
    }
```

the modifier `isListedNFT` will only allow code execution if an NFT is in fact listed.

However, the modifier logic does not take into account if the listed NFT has been canceled and thus really not listed

Recommendations

validate listedNFT.status does not equal canceled.

[L-07] Condition is checkInitialSale always evaluates to true

Description

The `GhostNFTMarketplace::checkInitialSale` function verifies if the NFT contract is the ghost NFT and if the id is from the initial sale. The initial sale includes only the first 10 NFTs e.i. from id 0 to 9 inclusive.

```
function checkInitialSale(
    address _nft,
    uint256 tokenId
) internal view returns (bool status) {
```

```
        status = _nft == GhostNFTAddress && (tokenId >= 10 || tokenId <= 150);
    }
```

The second condition however is wrong because it is always `true`. If the `tokenId` is 5 it is covered by `tokenId <= 150` and if it is above 150 it is covered by the `tokenId >= 10` condition

Recommendations

Require the id to be less than 10 as only the first 10 ids are from the initial sale

```
-       status = _nft == GhostNFTAddress && (tokenId >= 10 || tokenId <=
150);
+       status = _nft == GhostNFTAddress && tokenId < 10;
```

[L-08] Wrong age returned if NFT does NOT exist

Description

The protocol has a `public view` function that returns the age of the NFT in days and takes in the id of the NFT. So far the function is NOT used inside the protocol. The problem with it is that if the NFT does NOT exist the returned age would be non-zero since it would return the time sine timestamp in days because `birthTime` would be 0.

Recommendations

If the NFT `birthTime` is not set return 0 or revert.

```
    function getNFTAge(uint256 tokenId) public view returns (uint256) {
        NFT memory nft = nfts[tokenId];
+       if (nft.birthTime == 0) {
+          return 0;
+       }
        uint256 ageInSeconds = block.timestamp - nft.birthTime;
        return ageInSeconds / 86400; // Age in days
    }
```

[L-09] When auction ends winner is set to creator, not bidder

Description

When an auction ends the `winner` value in the `AuctionNFT` struct is updated. However, it is updated with the creator of the auction, not the winner. The NFT however is transferred to the actual highest bidder

which is as it should be. The winner value is NOT used in functionality afterward.

Recommendations

In the `GhostNFTMarketplace::resultAuction` update

```
-        auction.winner = auction.creator;
+        auction.winner = auction.lastBidder;
```

[L-10] Extra pay amount NOT refunded when buying NFT

Description

The protocol lets users list their NFTs for sale. When a user calls the `buyNFT` function there is a check that requires the `msg.value` to be equal or greater than the listing price. `require(msg.value >= listedNft.price, "Invalid price");` This is problematic since if a user overpays the overpaid amount is NOT refunded, unlike `Ghost_Raffle::mintTicket` and `GhostNFTMarketplace::initialSale` where the requirement is `msg.value` to equal (`==`) the price.

Recommendations

In the `GhostNFTMarketplace::buyNFT` function change:

```
-    require(msg.value >= listedNft.price, "Invalid price");
+    require(msg.value == listedNft.price, "Invalid price");
```

[L-11] Pay token is never used to pay for listed NFTs

Description

The protocol lets users list NFT for sale or auction them off. The seller can set a `payToken` when listing and it is checked if it is whitelisted. A token can only be whitelisted by the admin via the `addPayableToken` function and is then verified when put up for sale by the `isPayableToken` modifier. This is fine by itself, but every time an NFT is purchased the native token (`msg.value`) is used to pay and the pay token is ignored. It is only required to be passed in the list function and be whitelisted which is a big inconvenience for the seller. The case is the same for the buyer. In the `buyNFT` function it is required the passed pay token to be the same as in the listing, but the native token is used later instead.

```
    require(
        _payToken != address(0) && _payToken == listedNft.payToken,
        "Invalid pay token"
```

```
    );
...
    (bool sent, ) = payable(listedNft.seller).call{value: totalPrice}("");
```

Recommendations

Implement the paying with pay tokens functionality or remove it as a whole

[L-12] Off by one issue at `getValidAuctions`

Description

The `getValidAuctions` function returns all valid open auctions. The problem is that when it checks the auction end time it compares it against `block.timestamp` using the `>` operator. However, in the `bidAuction` function the `>=` operator is used.

Recommendations

Use `>=` instead of `>` when checking the end time in the `getAllValidAuctions` function so it corresponds with `bidAuction`.

[I-01] Wrong parameter set in mapping when a user creates an offer

Description

Always use a stable pragma to be certain that you deterministically compile the Solidity code to the same bytecode every time. The project is currently using multiple floatable versions - `pragma solidity ^0.8.20;` and `pragma solidity ^0.8.4;`. Furthermore, consider using a newer version of the compiler as the latest available version is 0.8.26

[I-02] Unused local variables

The the `GHOSTSBREEDING.sol` file inside the `NFT_ERC721` contract there are some variables that judging by the names are used for fees. However, they are never used besides having "getters" and "setters".

```
    uint256 private royaltyFee;
    address private royaltyRecipient;
...
    mapping(uint256 => address) public nftIndexToOwner;
```

If they are intended to tax NFT rent they should be implemented in the logic or removed

In the `GhostNFTMarketplace` contract: `mapping(uint256 => mapping(uint256 => mapping(address => uint256))) private bidPrices` is unused The `isNotListedNFT` and `isNotAuction` modifiers are unused The `checkInitialSale` function is unused

[I-03] Set revert message or use custom error

There is a require statement in `GeneScience::mixGenes` that lacks a revert message. This could confuse end users as to why the transaction has reverted and give no indication. Consider adding a revert message.

On the other hand, replacing all require statements with custom errors is also advised since using custom errors saves gas.

[I-04] Incorrect call of the ownable constructor

Severity

Impact: Low

Likelihood: Low

Description

In `MIDNIGHT.sol`

```
contract GhostNFTMarketplace is Ownable(msg.sender), ReentrancyGuard {
```

when declaring inheritance the protocol incorrectly adds msg.sender. I assume the protocol intended to do this in the constructor instead.

Recommendations

Add ownable in the constructor and pass in the msg.sender to correctly set the msg.sender to the owner of the contract. this is done correctly in the `GHOSTSBREEDING.sol` contract

```
    constructor(
        address geneScienceAddress,
        uint256 _royaltyFee,
        address _royaltyRecipient
    ) ERC721("Ghosts_Ashtray", "Ghost_NFT") Ownable(msg.sender) {
```