NextGen

Audited by Ethan Bennett

October 2023

Contents

About Darklinear Solutions 1

Introduction 1

Findings Summary 1

Findings 2

- Auction winner can cancel bid and receive refund while successfully claiming NFT . . . 2
- Minter can re-enter `NextGenMinterContract.mint` to exceed `maxAllowance` 3
- When a collection is configured to use `RandomizerNXT`, new mints can be previewed and conditionally reattempted in `onERC721Received` 4
- Missing checks for randomizer cause minting to fail without explanation if a randomizer has not been added for the collection 4
- Inadequate checks for empty values in `collectionPhases` (and `collectionData`) . . 4
- Some function-level admins can escalate their own roles to drain all funds from `NextGenMinterContract` 5
- Airdrop recipient can DoS airdrop by reverting in `onERC721Received` 5

About Darklinear Solutions

Darklinear Solutions provides unrivaled security for blockchain applications, from the bytecode to the browser. With years of experience in smart contract development and traditional software engineering, we find the bugs that others miss. Learn more at darklinear.com.

Introduction

NextGen is a platform for minting and auctioning generative art. This review consists of issues discovered during the course of Code4rena's competitive audit for NextGen in October 2023. It does not represent a full and exhaustive audit of the protocol.

The findings described below are classified according to Code4rena's standards.

Findings Summary

Classification	Finding
High risk (H-1)	Auction winner can cancel bid and receive refund while successfully claiming NFT
High risk (H-2)	Minter can re-enter `NextGenMinterContract.mint` to exceed `maxAllowance`
Low risk (L-1)	When a collection is configured to use `RandomizerNXT`, new mints can be previewed and conditionally reattempted in `onERC721Received`
Low risk (L-2)	Missing checks for randomizer cause minting to fail without explanation if a randomizer has not been added for the collection
Low risk (L-3)	Inadequate checks for empty values in `collectionPhases` (and `collectionData`)
Low risk (L-4)	[Some function-level admins can escalate their own roles to drain all funds from `NextGenMinterContract`
Low risk (L-5)	Airdrop recipient can DoS airdrop by reverting in `onERC721Received`

Findings

Auction winner can cancel bid and receive refund while successfully claiming NFT

Severity: High risk

Context: AuctionDemo.sol#L105, AuctionDemo.sol#L125

Description: This attack strings three smaller vulnerabilities together: reentrancy from `_safeTransfer`, off-by-one errors in the `auctionEndTime` checks in `auctionDemo.cancelBid` and `claimAuction`, and a failure to check that the low level refund calls in `AuctionDemo.claimAuction` succeed.

First, the attacker would need to pick a vulnerable collection: one with an `auctionEndTime` that coincides with a new `block.timestamp`. While `block.timestamp` is no longer vulnerable to miner manipulation, valid `block.timestamp` values can now be calculated well in advance. And with a new block reliably mined every twelve seconds, the attacker would likely not need to wait long for a vulnerable auction to occur.

Once an auction with a vulnerable `auctionEndTime` begins, the attacker would need to place a winning bid from a contract with a malicious `onERC721Received` function. With the high bid submitted, they would `claimAuction` at precisely the block coinciding with the `auctionEndTime`. This allows the attacker's malicious contract to call `cancelBid` in its `onERC721Received` callback.

While `claimAuction` checks that an auction has ended by requiring that `block.timestamp` is greater than or equal to the `auctionEndDate`, `cancelBid` checks that an auction has not ended by requiring that `block.timestamp` is less than or equal to the `auctionEndDate`. This means that, for exactly one second, the auction is both concluded and ongoing: a winning bid can be claimed and cancelled at the same time.

```
// auctionDemo.claimAuction L105
require(block.timestamp >= minter.getAuctionEndTime(_tokenid)
    && auctionClaim[_tokenid] == false
    && minter.getAuctionStatus(_tokenid) == true);

// auctionDemo.cancelBid L125
require(block.timestamp <= minter.getAuctionEndTime(_tokenid), "Auction ended");
```

Note that this attack is possible even without the reentrancy — this is just the more efficient path, and it is possibly cheaper as well (if the lower transaction costs adequately offset the cost of deploying the malicious contract, which is more likely to be the case if the attacker intends to exploit more than one auction).

Impact: While the auctioning account still gets paid, the last bidder in the array does not get their refund (unless the contract has the funds to cover the attacker's withdrawn bid). If there are multiple auctions happening at the same time, it is also possible that all refunds in the malicious transaction actually succeed, and the problem of the missing funds bubbles up in a later auction. In this scenario, the attack itself might not even be discovered until long after it occurs, if at all, and the attacker is free to continue exploiting vulnerable auctions as they come up.

Proof of Concept: View the full test here.

Recommended mitigation:

- Change L105 in `claimAuction` to:

```
require(block.timestamp > minter.getAuctionEndTime(_tokenid) && ( ... ) );
```

- Transfer the auctioned NFT to the winner after tranferring the winnings to the auctioner, or, better yet, after all other payments have been issued. This will adhere to the "checks, effects, interactions" pattern and prevent similar vulnerabilities from emerging here in the future.
- Be sure to `require(success)` for all low-level calls. This alone would have made it much more difficult to exploit an auction successfully.

Minter can re-enter `NextGenMinterContract.mint` to exceed `maxAllowance`

Severity: High risk

Context:

- NextGenCore.sol#L193

Description: Every collection has an explicitly defined `maxAllowance`. It will always need to be defined with some degree of thought, because failing to define it at all would give it the default value of 0 and disallow minting in most situations. Considering this, a user bypassing this limit and significantly exceeding the `maxAllowance` presents a significant problem.

Since `NextGenCore.mint` does not update the `tokensMintedPerAddress` until after it mints a new token, an attacker would only need to recursively re-enter `NextGenMinterContract.mint` in its `onERC721Received` function to mint as many tokens as they want. When the minting completes, the balances will update — but with the checks having been done on each loop before minting, the updated `tokensMintedPerAddress` would only matter if the attacker tried to mint again in a subsequent transaction.

Impact: The purpose of the novel approaches to minting in this project is to ensure fair and controlled mints for all users. Despite the attacker still paying the defined price for every NFT they purchase, they have still undermined this purpose. And for an extremely successful project in a healthy market, an attacker could gain more than enough ether to cover the initial costs and still profit in secondary markets.

The only upper limits on an attacker's ability to mint are the transaction's `gasLimit` and the collection's `totalSupply`.

Proof of Concept: View the full test here.

Recommended mitigation: Simply moving `_mintProcessing` (L193, `NextGenMinterContract.mint`) below the state updates would make this attack impossible.

When a collection is configured to use `RandomizerNXT`, new mints can be previewed and conditionally reattempted in `onERC721Received`

Severity: Low risk

Context: RandomizerNXT.sol#L58

Description: Since `RandomizerNXT` has already set the `tokenHash` by the time Core calls `_safeMint``, users could potentially preview some data about their mint in onERC721Received` before the transaction has completed, then revert and retry if desired. This is low risk, since this would all need to happen programmatically, and metadata and rarity-related attributes should not be available at this point in the minting process. It is possible that some collections might one day set data at the time of minting that can be advantageous to know in advance, though, so it should be patched regardless.

Missing checks for randomizer cause minting to fail without explanation if a randomizer has not been added for the collection

Severity: Low risk

Context: MinterContract.sol#L196

Description: Minting will not succeed unless a randomizer has been added for a collection. But adding a randomizer is only one of many configuration steps for a new collection, and it is separate from all other actions. It is an easy step to miss, but there is no validation that the randomizer has been set before minting. So in cases where it is forgotten, minting fails without a relevant error message.

Recommended mitigation: Consider requiring that a collection has a valid randomizer set before attempting to mint a new token.

Inadequate checks for empty values in `collectionPhases` (and `collectionData`)

Severity: Low risk

Context: MinterContract.sol#L165, NextGenCore.sol#L157

Description: On L164 of `NextGenMinterContract`, the function updates a boolean called `setMintingCosts` to `true`. It then checks the value of this boolean before it accesses essential values from the `collectionPhases` struct.

There is no validation that any of these parameters are non-null values, however, so `setMintingCosts` does not actually guarantee that any of these values exist. It then goes on to use `collectionPhases.timePeriod` — potentially `0` — as the denominater in the `getPrice` function. This would cause minting with sales option two to fail without a useful error message.

`collectionData` uses the same inadequate check, but none of its values would cause errors in critical functionality if they were zero.

Recommended mitigation: Consider adding additional requirements for the unchecked parameters in `setMintingCosts` and `core.setCollectionData`.

Some function-level admins can escalate their own roles to drain all funds from `NextGenMinterContract`

Severity: Low risk

Context: NextGenCore.sol#L322, MinterContract.sol#L461

Description: At the outset, it must be acknowledged that the NextGen team considers all admin roles to be trusted, and findings related to an admin abusing their defined role are not valid. However, an admin having the ability to escalate their own role unilaterally and execute functionality they were not approved to execute — even if only a malicious admin would exercise this ability — is worth fixing. This is even more true if this vulnerability is exploitable for major monetary gain, as is the case here.

Because the owner of the `NextGenAdmins` contract is treated as a global admin throughout the system, any `FunctionAdmin` with the authority to `updateAdminContract` (L454, `NextGenMinterContract`) can deploy their own admin contract, replace the old one, and drain all the funds from the minter contract by calling `emergencyWithdraw` (L461, `NextGenMinterContract`).

Even considering the team's intent to only use multisigs as admins, the balance of `NextGenMinter` could very easily get high enough to tempt more than one possible attacker. `emergencyWithdraw` is high-impact enough that it should not have any implicitly defined access granted to it. Such caveats are bound to be forgotten or overlooked, especially by future developers who join the NextGen team and external maintainers of forks (such as UNIC at launch).

Recommended mitigation: Consider explicitly defining who can access `updateAdminContract` and `emergencyWithdraw`, and under what circumstances. It would also be worthwhile to implement a multi-party approval system for these functions, as seen in `NextGenMinterContract` when proposing addresses and percentages.

Airdrop recipient can DoS airdrop by reverting in `onERC721Received`

Severity: Low risk

Context: NextGenCore.sol#L182

Description: Since `airdropTokens` iterates an array of airdrop `_recipients` and calls `_safeMint` for each, if any of these iterations reverts, the entire airdrop will fail. This makes it trivially easy for anyone who can secure a spot in an airdrop to execute a denial-of-service attack on the airdrop.

They would need to plan ahead by first deploying a contract with a malicious `onERC721Received` function, and then by doing whatever is required to qualify that contract for an airdrop. The difficulty of achieving this would depend on each collection's approach to putting their list of recipients together — based on similar projects, this could likely be achieved by sending other tokens by the same artist, or other NextGen tokens, to the contract until it is selected lottery-style (like The Memes).

Recommended mitigation: Consider using try/catch logic around each `_safeMint`, and emitting an event with relevant details when a mint fails. While not strictly necessary in other contexts, it would have the added benefit of providing the means to monitor failed mints of all kinds more closely.