CD SECURITY

AUDIT REPORT

Blessed
June 2024

Prepared by
immeas
minquanym

Introduction

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

Disclaimer

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

About **Blessed**

Blessed is a lottery project where the protocol will deploy a series of different lottery and auction contracts for a seller. The ticket buyers can then participate in these lotteries and auctions for the chance to win NFTs. The winner's deposits will be taken as payments for the NFT. Once the lottery has ended the seller can withdraw the proceeds minus a tax for the protocol.

Severity classification

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

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

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

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 3911a46766267faeb79f72dcb7e6bd9d412d120e**

Scope

The following smart contracts were in scope of the audit:

- `BlessedFactory.sol`

- `LotteryV1Base.sol`
- `LotteryV2Base.sol`
- `AuctionV1Base.sol`
- `AuctionV2Base.sol`
- `NFTTicketBase.sol`
- `Deposit.sol`

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

- Critical & High: 5 issues
- Medium: 7 issues
- Low: 6 issues

Findings Summary

ID	Title	Severity	Status
[C-01]	Any Winner can withdraw deposit after winning	Critical	Resolved
[C-02]	Winner can mint any number of NFTs using reentry	Critical	Resolved
[C-03]	User can bypass reroll & request randomness until win	Critical	Resolved
[H-01]	Participants can join knowing they'll win	High	Resolved
[H-02]	Buyer can request randomness multiple times	High	Resolved
[M-01]	`selectWinners()` incorrectly pops `eligibleParticipants`	Medium	Resolved
[M-02]	Users can exploit the system by depositing 1 wei	Medium	Resolved
[M-03]	User can join multiple times to increase chances of winning	Medium	Resolved
[M-04]	Attacker can spam function `deposit()` to always increase price	Medium	Resolved
[M-05]	Incorrect variable used in function `setupNewRound()`	Medium	Resolved
[M-06]	`startLottery()` sets status for the non-existing round	Medium	Resolved
[M-07]	Unbounded loop can prevent seller from withdrawing funds	Medium	Resolved
[L-01]	Unbounded loops can prevent lottery from starting	Low	Resolved
[L-02]	Handling multiple users bidding the same price	Low	Resolved
[L-03]	Implementation contract can be initialized	Low	Resolved
[L-04]	Avoid using `payable.transfer`	Low	Resolved
[L-05]	`fulfillRandomness()` call can fail	Low	Resolved

ID	Title	Severity	Status
[L-06]	Calls with unnecessary `payable` modifier	Low	Resolved
[I-01]	Missing events emitted on important state changes	Informational	Resolved
[I-02]	Code duplication	Informational	Resolved
[I-03]	Unused code	Informational	Resolved

Detailed Findings

[C-01] Winner can withdraw deposit after winning

Severity

Impact: High

Likelihood: High

Description

The lotteries in this protocol work in such a way that ticket buyers compete to have a chance to buy an NFT for the deposit they've provided. The proceeds from these NFTs can then be collected by the seller minus a protocol tax.

In `Deposit.sol` buyers buy tickets, then when the seller is satisfied with the amount of buyers, they start the lottery by calling `startLottery` followed by `initiateSelectWinner`. This will request randomness from the VRF. Once randomness is returned the seller calls `selectWinners` where the winners will be selected using the randomness provided.

The issue is that during the time between that the randomness is fulfilled in `fulfillRandomWords` and the seller calls `selectWinners` the buyer will know if they've won and can withdraw their deposit.

Since the buyer is already added to the `eligibleParticipants` from when `startLottery` was called. They would still be able to win. Hence the buyer would get the NFT without paying.

PoC

Add this test to `Deposit.t.sol`:

```
    function test_WinnerWithdrawAfterRandomnessIsFulfulled() public {
        address winner = address(3);
        uint256 depositAmount = 1 ether;

        vm.deal(winner, depositAmount);
        vm.prank(winner);
        deposit.deposit{value: depositAmount}();
```

```
        // seller starts lottery
        vm.startPrank(seller);
        deposit.setNumberOfTickets(1);
        deposit.startLottery();
        uint256 requestId = deposit.initiateSelectWinner();
        vm.stopPrank();

        // VRF returns with randomness
        vrfMock.fulfillRandomWords(requestId, address(deposit));

        // winner sees that they won and withdraws their deposit
        vm.prank(winner);
        deposit.buyerWithdraw();

        // seller sets winner
        vm.startPrank(seller);
        deposit.selectWinners();
        deposit.endLottery();
        deposit.sellerWithdraw();
        vm.stopPrank();

        // winner both won and have their deposit
        assertEq(winner.balance,1 ether);
        assertEq(deposit.isWinner(winner),true);

        // seller got nothing
        assertEq(seller.balance,0);
    }
```

Recommendations

Consider only letting buyers withdraw after the lottery has ended:

```
-   function buyerWithdraw() public whenLotteryNotActive {
+   function buyerWithdraw() public lotteryEnded {
```

[C-02] Winner can mint any number of NFTs using reentry

Severity

Impact: High

Likelihood: High

Description

When someone wins in `AuctionV1` they can mint an NFT:

```
    function mintMyNFT() public {
        require(isWinner(_msgSender()), "Caller is not a winner");
        require(!hasMinted[_msgSender()], "NFT already minted");
        INFTLotteryTicket(nftContractAddr).lotteryMint(_msgSender());
        hasMinted[_msgSender()] = true;
    }
```

The issue is that this is susceptible to reentry since the `hasMinted` state is changed after the call to `lotteryMint`. `lotteryMint` will mint an `ERC1155` token which, if the receiver is a contract, calls `onERC1155Received` on the receiver. This can be used to reenter into `mintMyNFT` to mint any number of NFTs.

PoC

```
    function test_WinnerReentrancyToMintMoreNfts() public {
        AuctionWinner winner = new AuctionWinner(usdc, auction);

        usdc.mint(address(winner), 1e6);

        winner.deposit();

        vm.startPrank(seller);
        auction.startLottery();
        auction.setupNewRound(block.timestamp, 1);
        auction.selectWinners();
        vm.stopPrank();

        winner.mintNft();

        assertEq(nft.balanceOf(address(winner), 1), 10);
    }
```

Using this code in the reentry contract:

```
    function mintNft() public {
        auction.mintMyNFT();
    }

    function onERC1155Received(address, address, uint256, uint256, bytes
calldata) public returns(bytes4) {
        uint256 balance =
NFTTicketBase(auction.nftContractAddr()).balanceOf(address(this),1);
        // abuse reentry to mint 10 nfts
        if(balance < 10) {
            auction.mintMyNFT();
        }
```

```
        return IERC1155Receiver.onERC1155Received.selector;
    }
```

Please find the whole test setup here.

Recommendations

Consider changing the `hasMinted` before the call to mint:

```
    function mintMyNFT() public {
        require(isWinner(_msgSender()), "Caller is not a winner");
        require(!hasMinted[_msgSender()], "NFT already minted");
+       hasMinted[_msgSender()] = true;
        INFTLotteryTicket(nftContractAddr).lotteryMint(_msgSender());
-       hasMinted[_msgSender()] = true;
    }
```

[C-03] User can bypass reroll & request randomness until win

Severity

Impact: High

Likelihood: High

Description

To participate in LotteryV2 the ticket buyer calls `LotteryV2Base::deposit`. This will trigger a randomness request to the VRF:

```
147:    function deposit(uint256 amount) public lotteryStarted
hasNotWonInLotteryV1(_msgSender()) {
148:        require(usdcContractAddr != address(0), "USDC contract address
not set");
149:        require(amount > 0, "No funds sent");
...
162:        if (rolledNumbers[_msgSender()] == 0) {
163:            _requestRandomness(abi.encode(_msgSender()));
164:            emit RandomRequested(_msgSender());
165:        }
166:    }
```

This randomness request is later processed by the VRF in `LotteryV2Base::_fulfillRandomness`, where on line 142 the number is checked against the winning number:

```
134:    function _fulfillRandomness(uint256 randomness, uint256, bytes
memory extraData) internal override {
135:        address requestedBy = abi.decode(extraData, (address));
136:        uint256 _randomNumber =
DigitExtractor.extractFirst14Digits(randomness);
137:
138:        if (requestedBy == seller) {
139:            randomNumber = _randomNumber;
140:        } else {
141:            rolledNumbers[requestedBy] = _randomNumber;
142:            claimNumber(requestedBy);
143:        }
144:        emit RandomFullfiled(requestedBy, _randomNumber);
145:    }
```

If the user wants a second roll they can call `roll` and pay a fee to roll again.

The issue is that this isn't the only way. There is an unprotected call `requestRandomness`:

```
129:   function requestRandomness() external {
130:        _requestRandomness(abi.encode(_msgSender()));
131:        emit RandomRequested(_msgSender());
132:    }
```

Using this a buyer can roll as many times as they want until they get a winning number.

PoC

```
    function test_LotteryV2RequestRequestRandomnessMultipleTimesReroll()
public {
        usdc.mint(alice, 2e6);

        vm.startPrank(alice);
        usdc.approve(address(lottery), 1e6);

        bytes memory data = abi.encode(0, abi.encode(alice));
        uint256 round = _round();
        bytes memory dataWithRound = abi.encode(round, data);

        vm.expectEmit(false, false, false, true, address(lottery));
        emit IGelatoVRFConsumer.RequestedRandomness(round, data);

        lottery.deposit(1e6);
        vm.stopPrank();

        lottery.mockFulfillRandomness(12345678901234, alice);
        assertEq(lottery.rolledNumbers(alice), 12345678901234);
```

```
        data = abi.encode(1, abi.encode(alice));
        dataWithRound = abi.encode(round, data);
        vm.expectEmit(false, false, false, true, address(lottery));
        emit IGelatoVRFConsumer.RequestedRandomness(round, data);

        // alice can call `requestRandomness` multiple times
        // even after randomness is fulfilled
        vm.prank(alice);
        lottery.requestRandomness();

        data = abi.encode(2, abi.encode(alice));
        dataWithRound = abi.encode(round, data);
        vm.expectEmit(false, false, false, true, address(lottery));
        emit IGelatoVRFConsumer.RequestedRandomness(round, data);

        vm.prank(alice);
        lottery.requestRandomness();
    }
```

Please find the full test setup here

Recommendations

Consider adding `onlySeller` to the `requestRandomness`.

[H-01] Participants can join knowing they'll win

Severity

Impact: Medium

Likelihood: High

Description

Winners are picked in `LotteryV1` and `AuctionV1` by the seller calling `selectWinners`. This selects `eligibleParticipants`. Given that the number of eligible participants is bigger than the number of tickets, the participants are shuffled using the randomness provided by the VRF:

```
196:     function selectWinners() external onlySeller lotteryStarted {
197:         require(numberOfTickets > 0, "No tickets left to allocate");
198:         checkEligibleParticipants();
199:
200:         if(numberOfTickets >= eligibleParticipants.length) {
                 // ... everyone wins
212:         } else {
213:             // Shuffle the array of participants
214:             for (uint j = 0; j < eligibleParticipants.length; j++) {
215:                 uint n = j + randomNumber %
(eligibleParticipants.length - j);
```

```
216:                 address temp = eligibleParticipants[n];
217:                 eligibleParticipants[n] = eligibleParticipants[j];
218:                 eligibleParticipants[j] = temp;
219:             }
220:
221:             // Select the first `numberOfTickets` winners
```

The issue is that the random number has to be provided *before* this is called. Since the random number is known to everyone a user can run the above algorithm and see which spots in `eligibleParticipants` that wins. More importantly, if the last slot would win, if the list was increased by one from them joining (or two with them joining twice and so on).

Thus after randomness is selected but before `selectWinners` is called they can join knowing if they'll win or not. This applies to everyone, but the last one in will always have greater knowledge than all previous participants which is unfair.

Since they buyer can withdraw their deposit if they don't win this exploit costs only gas for the exploiter.

Recommendations

Consider requiring deposits to be done before the lottery starts:

```
-   function deposit(uint256 amount) public payable lotteryStarted {
+   function deposit(uint256 amount) public payable lotteryNotStarted {
```

[H-02] Buyer can request randomness multiple times

Severity

Impact: Medium

Likelihood: High

Description

To participate in LotteryV2 the ticket buyer calls `LotteryV2Base::deposit`. This will trigger a randomness request to the VRF:

```
147:    function deposit(uint256 amount) public lotteryStarted
hasNotWonInLotteryV1(_msgSender()) {
148:        require(usdcContractAddr != address(0), "USDC contract address
not set");
149:        require(amount > 0, "No funds sent");
...
162:        if (rolledNumbers[_msgSender()] == 0) {
163:            _requestRandomness(abi.encode(_msgSender()));
164:            emit RandomRequested(_msgSender());
```

```
165:         }
166:     }
```

This randomness request is later processed by the VRF in `LotteryV2Base::_fulfillRandomness`, where on line 142 the number is checked against the winning number:

```
134:     function _fulfillRandomness(uint256 randomness, uint256, bytes
memory extraData) internal override {
135:         address requestedBy = abi.decode(extraData, (address));
136:         uint256 _randomNumber =
DigitExtractor.extractFirst14Digits(randomness);
137:
138:         if (requestedBy == seller) {
139:             randomNumber = _randomNumber;
140:         } else {
141:             rolledNumbers[requestedBy] = _randomNumber;
142:             claimNumber(requestedBy);
143:         }
144:         emit RandomFullfiled(requestedBy, _randomNumber);
145:     }
```

The issue here is that in `deposit` it only checks that the user doesn't have a *fulfilled* request. VRFs take some time to process requests. A ticket buyer could send as many requests as they can before the first one is fulfilled which will increase their changes to win. Since as long as one of the rolls succeed they'll get added to the `winners` list.

Only the first one need to be above the minimum deposit threshold, the later ones can just be dust. Hence a buyer could get many rolls for just one ticket price.

PoC

```
    function test_LotteryV2RequestRandomnessMultipleTimesDeposit() public
{
        usdc.mint(alice, 2e6);

        vm.startPrank(alice);
        usdc.approve(address(lottery), 2e6);

        // first randomness request with enough to cover minimum deposit
        bytes memory data = abi.encode(0, abi.encode(alice));
        uint256 round = _round();
        bytes memory dataWithRound = abi.encode(round, data);

        vm.expectEmit(false, false, false, true, address(lottery));
        emit IGelatoVRFConsumer.RequestedRandomness(round, data);

        lottery.deposit(1e6);

        // second randomness request while first is still pending for dust
```

```
        data = abi.encode(1, abi.encode(alice));
        dataWithRound = abi.encode(round, data);

        vm.expectEmit(false, false, false, true, address(lottery));
        emit IGelatoVRFConsumer.RequestedRandomness(round, data);

        lottery.deposit(1);

        vm.stopPrank();
    }
```

Please find the full test setup here

Recommendations

Consider only sending a randomness request if both `rolledNumbers[_msgSender()] == 0` and `deposits[_msgSender()] == 0` in both `LotteryV2Base::deposit` and `transferDeposit`:

```
        if (deposits[_msgSender()] == 0) {
            participants.push(_msgSender());

+           if (rolledNumbers[_msgSender()] == 0) {
+               _requestRandomness(abi.encode(_msgSender()));
+               emit RandomRequested(_msgSender());
+           }
        }
        deposits[_msgSender()] += amount;

-       if (rolledNumbers[_msgSender()] == 0) {
-           _requestRandomness(abi.encode(_msgSender()));
-           emit RandomRequested(_msgSender());
-       }
```

This will guarantee that only one request is sent per ticket buyer.

[M-01] `selectWinners()` incorrectly pops `eligibleParticipants`

Severity

Impact: Low

Likelihood: High

Description

In the `selectWinners()` function, when demand exceeds supply, only some participants in the `eligibleParticipants` list will receive a ticket. The function then removes the selected winners,

retaining only the other participants. It achieves this by shifting the index and popping out from the back of the list.

```
// Select the first `numberOfTickets` winners
for (uint256 i = 0; i < numberOfTickets; i++) {
    address selectedWinner = eligibleParticipants[i];
    if (!isWinner(selectedWinner)) {
        setWinner(selectedWinner);
        emit WinnerSelected(selectedWinner);
    }
}

// Remove the winners from the participants list by shifting non-winners
up
uint256 shiftIndex = 0;
for (uint256 i = numberOfTickets; i < eligibleParticipants.length; i++) {
    eligibleParticipants[shiftIndex] = eligibleParticipants[i];
    shiftIndex++;
}
// @audit `eligibleParticipants.length` will decrease after pop()
for (uint256 i = shiftIndex; i < eligibleParticipants.length; i++) {
    eligibleParticipants.pop();
}
```

However, the final loop that pops out elements uses `i < eligibleParticipants.length` as the stopping condition, even though the `eligibleParticipants` list is constantly updated during the process.

As a result, the `eligibleParticipants.length` decreases after every `pop()` operation, leading to incorrect function results.

For instance, let's say we have a list `eligibleParticipants = [Alice, Bob]` and `shiftIndex = 0`. The function tries to pop out `Alice` and `Bob`.

- At `i = 0`, `Bob` is popped out, making `eligibleParticipants = [Alice]`.
- At `i = 1`, the function breaks since `eligibleParticipants.length == i == 1` now.

Recommendations

Rather than using `eligibleParticipants.length` while popping out values, cache the length before the loop.

[M-02] Users can exploit the system by depositing 1 wei

Severity

Impact: Medium

Likelihood: Medium

Description

In the `LotteryV2Base` contract, buyers can roll a dice to generate a random number. Those who roll numbers close to the seller's number become eligible for minting.

Each roll requires a `rollPrice`. However, users receive a "free" roll in the `deposit()` function. This can be exploited by an attacker. The attacker might only deposit 1 wei into the contract to get this "free" roll, and then deposit the necessary amount to claim the number if they roll a winning number. If they don't roll a winning number, they can simply create and deposit into a new account/address.

```
function deposit(uint256 amount) public lotteryStarted
hasNotWonInLotteryV1(_msgSender()) {
    require(usdcContractAddr != address(0), "USDC contract address not
set");
    require(amount > 0, "No funds sent");
    require(
        IERC20(usdcContractAddr).allowance(_msgSender(), address(this)) >=
amount,
        "Insufficient allowance"
    );

    IERC20(usdcContractAddr).transferFrom(_msgSender(), address(this),
amount);

    if (deposits[_msgSender()] == 0) {
        participants.push(_msgSender());
    }
    deposits[_msgSender()] += amount;

    // @audit Free roll
    if (rolledNumbers[_msgSender()] == 0) {
        _requestRandomness(abi.encode(_msgSender()));
        emit RandomRequested(_msgSender());
    }
}
```

Recommendations

Consider charging the `rollPrice` in the `deposit()` function.

[M-03] User can join multiple times to increase chances of winning

Severity

Impact: Medium

Likelihood: Medium

Description

When joining the lottery in `Deposit` a user would call `deposit`:

```
90:     function deposit() public payable whenLotteryNotActive {
91:         require(msg.value > 0, "No funds sent");
92:         if (deposits[msg.sender] == 0) {
93:             participants.push(msg.sender);
94:         }
95:
96:         deposits[msg.sender] += msg.value;
97:     }
```

Once a participant is in the `participants` list, if they have a large enough deposit they will be added to `eligibleParticipants` in `checkEligibleParticipants`, when the seller has called `startLottery`:

```
225:     function checkEligibleParticipants() internal {
226:         for (uint256 i = 0; i < participants.length; i++) {
227:             uint256 depositedAmount = deposits[participants[i]];
228:             if (depositedAmount >= minimumDepositAmount) {
229:                 // Mark this participant as eligible for the lottery
230:                 eligibleParticipants.push(participants[i]);
231:             }
232:         }
233:     }
```

The issue here is that the buyer can add themselves multiple times to `participants`. The buyer can call `deposit` followed by `withdrawDeposit` then `deposit` again. Since withdrawing the deposit sets your `deposits` to `0` the second call to `deposit` would add them again in the `participants` list. This still just having paid for one deposit.

Which later would grant them multiple spots in `eligibleParticipants` from where the winner is picked from.

The user would still just win once since there's a check in `selectWinners` that they haven't won before.

Note, technically, since a user can withdraw the funds if they didn't win, there could be a possibility of a sybil attack to win a lot of the NFTs. But since each of these would come from a separate account they would all have the same "expected cost" since they'll have to pay as many times as they won. This is not the case here, this is just increasing the likelihood of one account winning with no increased cost.

PoC

Add this test to `Deposit.t.sol`:

```
    function test_DepositJoinMultipleTimes() public {
        address user = address(3);
        vm.deal(user, 1e18);

        vm.startPrank(user);
        deposit.deposit{value: 1e18}();

        deposit.buyerWithdraw();

        deposit.deposit{value: 1e18}();
        vm.stopPrank();

        address[] memory participants = deposit.getParticipants();
        assertEq(participants[0],user);
        assertEq(participants[1],user);
    }
```

Recommendations

Consider only letting buyers withdraw after the lottery has ended:

```
-   function buyerWithdraw() public whenLotteryNotActive {
+   function buyerWithdraw() public lotteryEnded {
```

[M-04] Attacker can spam function `deposit()` to always increase price

Severity

Impact: Medium

Likelihood: Medium

Description

In `AuctionV1Base`, there are multiple rounds with fluctuating prices, depending on user demand. User demand is tracked by the `prevRoundDeposits` variable, which increases each time a user makes a deposit.

```
function deposit(uint256 amount) public payable {
    ...

    if(deposits[_msgSender()] == 0) {
        participants.push(_msgSender());
    }
    deposits[_msgSender()] += amount;
```

```
    // @audit Attacker can spam to fake demand and always increase price
    prevRoundDeposits += 1;
}
```

However, the `deposit()` function can be called to deposit as little as 1 wei, yet the `prevRoundDeposits` still increases. An attacker could exploit this by repeatedly calling `deposit()`, creating the illusion of high demand and driving prices up. Even without buying a ticket, they could force others to pay more.

Recommendations

Consider adjusting the code to only increase `prevRoundDeposits` when users deposit an amount equal to or greater than the current ticket price.

[M-05] Incorrect variable used in function `setupNewRound()`

Severity

Impact: Medium

Likelihood: Medium

Description

The process of a round in AuctionV1Base is as follows:

1. The operator invokes the `setupNewRound()` function to calculate the `newPrice` and start a new round. The state of the new round is set to `lotteryStarted = false` and `winnersSelected = false`.
2. The seller invokes the `startLottery()` function, setting the round state to `lotteryStarted = true`.
3. The seller calls the `selectWinner()` function to conclude the round and set `winnersSelected = true`.

In the `setupNewRound()` function, the `numberOfTickets` variable is used to determine whether the price should increase or decrease. However, `numberOfTickets` always resets to `0` when the previous round ends in the `selectWinner()` function.

```
function setupNewRound(uint256 _finishAt, uint256 _numberOfTickets) public
onlyOperator {
    require(_numberOfTickets <= totalNumberOfTickets, "Tickets per round
cannot be higher than total number of tickets in AuctionV1");
    uint256 newPrice = 0;

    // @audit `numberOfTickets` is always reset to `0`
    //         after finishing previous round in `selectWinner()`
    if (prevRoundDeposits >= numberOfTickets) {
      ...
```

```
    }
    ...
}
```

Recommendations

Consider using `prevRoundTicketsAmount` instead of `numberOfTickets` as that is kept constant after round end:

```
-       if (prevRoundDeposits >= numberOfTickets) {
+       if (prevRoundDeposits >= prevRoundTicketsAmount) {
```

[M-06] `startLottery()` sets status for the non-existing round

Severity

Impact: Medium

Likelihood: Medium

Description

In `AuctionV1Base`, the `rounds` mapping uses a 0-index. This means the valid indices range from `[0, roundCounter - 1]`. As a result, the function `startLottery()` sets the `lotteryStarted` status for a non-existing round.

```
function startLottery() public onlySeller lotteryNotStarted {
    changeLotteryState(LotteryState.ACTIVE);
    checkEligibleParticipants();
    // @audit Set status for the wrong round
    //        `roundCounter` is not existed
    rounds[roundCounter].lotteryStarted = true;
}
```

Recommendations

Consider changing the index to `roundCounter - 1`.

[M-07] Unbounded loop can prevent seller from withdrawing funds

Severity

Impact: High

Likelihood: Low

Description

In `LotteryV2`, the winner is picked from how close their random number is to the sellers random number.

When a buyer has won, their win is automatically claimed from the VRF call to `fulfillRandomness`:

```
140:         } else {
141:             rolledNumbers[requestedBy] = _randomNumber;
142:             claimNumber(requestedBy);
143:         }
```

And in `claimNumber`:

```
288:     function claimNumber(address _participant) public returns (bool) {
289:         if (isClaimable(_participant)) {
290:             winners[_participant] = true;
291:             winnerAddresses.push(_participant);
292:             emit WinnerSelected(_participant);
293:             return true;
294:         } else {
295:             return false;
296:         }
297:     }
```

`winnerAddresses` is then iterated over and each winners deposit is added up and sent (after taking protocol tax) to the seller in `sellerWithdraw`.

The issue is that `claimNumber` can be called by anyone any number of times. Each call the winner is added to `winnerAddresses` again. This could be used repeatedly to make the list so large that the gas for the transaction would not fit in a block.

Recommendations

Consider checking if a user has already claimed before adding to `winnerAddresses`:

```
-       if (isClaimable(_participant)) {
+       if (isClaimable(_participant) && !winners[_participant]) {
```

[L-01] Unbounded loops can prevent lottery from starting

`Deposit`, `LotteryV1`, `AuctionV1` all store the participants in an array `participants`. Then when the lottery is supposed to start this array is iterated over and the users with enough deposit are added to an array `eligibleParticipants`. `LotteryV1Base::checkEligibleParticipants`:

```
273:     function checkEligibleParticipants() internal {
274:         for (uint256 i = 0; i < participants.length; i++) {
275:             uint256 depositedAmount = deposits[participants[i]];
276:             if (depositedAmount >= minimumDepositAmount) {
277:                 // Mark this participant as eligible for the lottery
278:                 eligibleParticipants.push(participants[i]);
279:             }
280:         }
281:     }
```

This looks similar in the other contracts.

The issue here is that there is no cost to adding participants. A griefer could add enough accounts to cause the calls that involve `checkEligibleParticipants` to run out of gas. This preventing the lottery from starting. This would never put any funds at risk as the buyers can always withdraw their deposits but it can prevent the lottery from proceeding.

We recommend you enforce the `minumumDepositAmount` in the `deposit` call instead.

[L-02] Handling multiple users bidding the same price

The `AuctionV2Base` contract currently does not address the situation where multiple users bid the same price. At present, it merely sorts all the deposits by amount. If the amounts are identical, any order is considered valid. This could result in unfairness for buyers who bid the same price but did not secure the ticket.

```
// @audit should sort by timestamp or sth else when amounts are equal
function sortDepositsDesc() public onlySeller {
  for (uint256 i = 0; i < participants.length; i++) {
    for (uint256 j = i + 1; j < participants.length; j++) {
      if (deposits[participants[i]].amount <
deposits[participants[j]].amount) {
        address temp = participants[i];
        participants[i] = participants[j];
        participants[j] = temp;
      }
    }
  }
}
```

Consider revising the mechanism to ensure fairness for all users. For instance, deem all participants with the same winning price as winners. Alternatively, consider adding timestamps when sorting, to prioritize

users who bid first.

[L-03] Implementation contract can be initialized

`LotteryV1Base`, `LotteryV2Base`, `AuctionV1Base`, `AuctionV2Base`, `NFTTicketBase` are all intended to be used as proxies (clones).

The issue is that these implementation contracts can be initialized themselves. Which can be confusing even though not immediately dangerous.

We recommend you prevent the implementation contracts from being able to be initialized. Either by adding a constructor that sets `initialized` to `true` or by extending OpenZeppelin `Initializable` and using the tools from there.

[L-04] Avoid using `payable.transfer`

In `Deposit.sol` `payable.transfer` is used to send `eth` to different parties:

```
135:         payable(msg.sender).transfer(amount);
...
152:         payable(multisigWalletAddress).transfer(protocolTax);
153:         payable(seller).transfer(amountToSeller);
```

`payable.transfer` only forwards a fixed 2300 gas for the call. This could cause multisigs and other smart wallets to run out of gas since they do logic in their `receive` functions.

Consider using low-level `call.value(amount)` with the corresponding result check or using the OpenZeppelin library `Address.sendValue`. As all of the above calls are done after any state changes are done there is no risk for reentrancy.

[L-05] `fulfillRandomness()` call can fail

It is best practice to make sure the call to `fulfillRandomness` cannot revert. Since otherwise if the call can revert they wouldn't be able to provide their randomness.

This is not the case in `LotteryV1Base::_fulfillRandomness`:

```
134:     function _fulfillRandomness(uint256 randomness, uint256, bytes
memory extraData) internal override {
135:         address requestedBy = abi.decode(extraData, (address));
136:         uint256 _randomNumber =
DigitExtractor.extractFirst14Digits(randomness);
```

The function `extractFirst14Digits` will revert if `randomness` is not 14 or more digits. Hence if the roll is `<10_000_000_000_000` the call will revert.

Consider using the last 14 digits instead of the first:

```
-       uint256 _randomNumber =
DigitExtractor.extractFirst14Digits(randomness);
+       uint256 _randomNumber = randomness % 100_000_000_000_000; // last
14 digits
```

Since all the 256 bits have the same entropy there is no difference in the randomness between the highest and the lowest bits in the number.

[L-06] Calls with unnecessary `payable` modifier

Some calls have a `payable` modifier without having a need to receive ETH:

- `LotteryV1Base::deposit`
- `AuctionV1Base::deposit`
- `AuctionV2Base::deposit`

This can cause users to accidentally send eth to the contract.

Consider removing payable from the above calls.

[I-01] Missing events emitted on important state changes

Events are important for off-chain tracking of what is happening on-chain and it's good practice to emit events for any important state changes to the contract. A lot of calls throughout the code base are missing events when important states are changed. Anything that changes the state should emit an event so that this can be tracked and monitored off-chain.

Consider adding events to all calls that change state throughout the code base

[I-02] Code duplication

Throughout the code base there is a lot of code duplication. For example, the progression of states within the lotteries/auctions is shared across all contracts.

The same goes for selection of winners and transfers of tickets to subsequent stages.

Consider reviewing where common code is used and move this to libraries or common base contracts. This will increase code readability and ease the development since code will be reused instead of having to remember to copy an implementation to everywhere it is used.

[I-03] Unused code

There are some unused functions and variables in the codebase:

- Functions:
- Internal `removeParticipant()` functions.
- Public view `getRandomNumber()` function in `AuctionV1Base`.
- Enum `LotteryState`: `VRF_REQUESTED` and `VRF_COMPLETED`.
- The `config._seller` variable is unused, and the `seller` variable is assigned `config._blessedOperator` instead.

Consider removing the unused code.