QuillAudits

Audit Report
August, 2022

For

yolorekt

Table of Content

Executive Summary

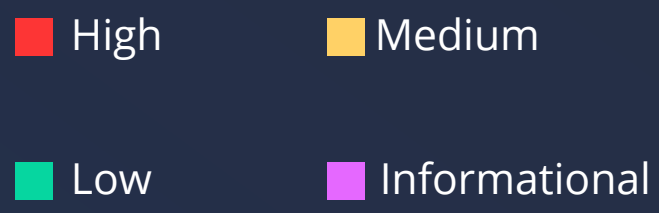
Project Name	*YoloRekt*
Overview	YOLOrekt is a decentralized short-term prediction exchange. It allows users to predict whether the final price will end up above or below a strike price and place bids on the outcome. Liquidity on YOLOrekt is provided by decentralized liquidity pools that are backed by liquidity providers. In order to enable decentralized liquidity provider (LP) pools, YOLOrekt uses a utility token called YOLO. This allows YOLOrekt to solve a number of issues concerning in-game liquidity and incentivizes liquidity providers by allowing LPs to stake YOLO tokens directly into the game liquidity pool and lock in tokens for additional rewards and privileges.
Timeline	March 7th, 2022 to July 20, 2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyze YoloRekt's codebase for quality, security, and correctness. *YoloRekt's CodeBase* Commit: 9762afc183c7f2f4b1e84d1d835767c74faa69fb *BiddersRewards* Commit: cea8d30406d814fd5882c16207b83461ac154313 *BiddersRewardsFactory* Commit: cea8d30406d814fd5882c16207b83461ac154313
Fixed In	1f31ff769af9ebfe3b920daf5e65afd31fa3c19f

Executive Summary

YoloRegistry	Registry and controller contract which keeps track of critical yolo contracts info, including latest contract addresses and version
Users	pools user token deposits into Yolo market system
RegistrySatellite	Base contract for all Yolo contracts that depend upon YoloRegistry for references of other contracts (particularly their active addresses), supported assets (and their token addresses if applicable), registered game contracts, and master admins
NFTTracker	Tracks bids made by participants, level requirement thresholds, NFT data.
LiquidityPool	Provide liquidity for the game rounds by depositing YOLO token and receiving an LP share token in return (which can be deposited in the Staking Rewards contract)
CoreCommon	Base contract to RegistrySatellite, used to restrict critical method calls to admin only
IYoloGame	Basic interface to Yolo Game contracts
GameInstance	A binary prediction market for a given asset pair, `gamePair`, and round duration denoted in a number of seconds, `gameLength`. Manages game round starts and settlements, and handles bids executed by users
GameFactoryWithNFTPack	The factory is in charge of minting new game instances for a given pair and game length (block duration). Already existing doubles will revert.
GameInstanceWithNFTPack	A wrapper on GameInstance contract in order to call the NFTTracker contract, which keeps track of rounds and cumulative amount bidder has participated - used to earn bidder rewards.
BiddersRewards	Reward participants vis-a-vis nft linked data on bid count and cumulative amount bid

Executive Summary

BiddersRewardsFactory	Factory to deploy/rotate BiddersRewards contracts
YoloNFTPack	A wrapper around custom Yolo ERC1155 extensions with functions for creating participation tokens for members, allowing users to create a base nft token and upgrade qualified tokens to higher levels or tranches.
ERC1155DynamicURI	A bypass for {IERC1155MetadataURI} with custom setURI quasi function overload for dynamic id-specific URIs, in order to provide long term support for IPFS CIDs.
ERC1155SemiFungible	Modification of {ERC1155MixedFungible} to provide semi-fungible (SFT) and NFT support in split bit compact form w/ max balance of 1 for each SFT series per address. SFT base types will all share the same metadata uri.
YoloEthereumUtilityTokens	standard ERC20 fixed-supply contract.
YoloShareTokens	Mintable LP share tokens produced when liquidity providers deposit YOLO token in the LiquidityPool contract
SplitBitId	Bit masking library for encoding and decoding SFT and NFT token ids
constants	Constant identifier values



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	**5**	**5**	**7**	**6**
Partially Resolved Issues	0	0	0	0
Resolved Issues	**10**	**5**	**4**	**6**

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. SplitBitId

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

A1. Encoding New BaseTypes may lead to collision/overlapping with existing BaseTypes

```
61      function encodeNewNonFungibleBaseType(uint256 _rawNonce)
62          internal
63          pure
64          returns (uint256)
65      {
66          return (_rawNonce << 128) | TYPE_NF_BIT;
67      }
68
69      function encodeNewSemiFungibleBaseType(uint256 _rawNonce)
70          internal
71          pure
72          returns (uint256)
73      {
74          return (_rawNonce << 128) | TYPE_SEMI_BIT;
75      }
```

Functions encodeNewNonFungibleBaseType and encodeNewSemiFungibleBaseType allows encoding new basetypes. The functions expect and allow a uint256(0 - 2^256-1) value to be encoded. However, the maximum value that can be encoded or the last available nonce that can be encoded as a basetype is 2^126-1, which means values >=2^126, will start overlapping/ colliding with existing baseTypes.

Example Scenario

Encoding rawNonce 2^126 as new SemiFungible BaseType, will collide with baseType 0 or TYPE_SEMI_BIT
Encoding rawNonce 2^126 as new NonFungible BaseType, will collide with baseType 0 or TYPE_SEMI_BIT, i.e encoding a rawNonce >=2^126 as new NonFungible BaseType will actually produce a SemiFungible BaseType

Recommendation

Consider adding check to allow rowNonces to not exceed 2^126-1 for new SemiFungible or NonFungible BaseTypes

Status

Acknowledged

Comment: The team said that they were skipping this as it is “unreachable” because all index and baseType nonces are incremented by one

Informational Issues

No issues found

B. YoloRegistry

High Severity Issues

No issues found

Medium Severity Issues

B1. Missing Address Validation

```
82      function setContract(bytes32 identifier, ContractDetails calldata newData)
83          external
84          onlyAdmin
85      {
86          ContractDetails storage oldRegister = contractRegistry[identifier];
87
88          require(
89              newData.version == oldRegister.version + 1,
90              "new version val must be 1 g.t."
91          );
92
93          address oldAddress = oldRegister.contractAddress;
94
```

Function allows Admins to register a new contract to the Yolo Ecosystem. However, there exists no input validation for the contract address to be registered, as a result, address collisions may happen, or in other words, the same address can be registered for multiple contract identifiers.

Example Scenario

Registering, the same address for two different contract identifiers for USERS and YOLO_TOKEN, which may lead to unexpected behavior, as all yolo ecosystem's contracts rely upon the yoloRegistry for the contract initializations and other references/operations.

Recommendation

Consider adding the required checks in order to reduce the risks of incorrect contract initializations and operations.

Status

Acknowledged

Dev Comment: Added validation for existing contracts before registering 2f2a7cd094e66f981f992160e44a1309517d55d6

There is also a draft with suggested additional checks - *Read here*

Low Severity Issues

No issues found

Informational Issues

B2. Unused Code

```
23      mapping(bytes32 => ContractDetails) contractRegistry;
24      // game instances preapproved for factory minting
25      mapping(address => bool) public registeredGames;
26      // approved token contracts for use in system - key can be keccak hash of symbol, e.g., "WETH"
27      mapping(bytes32 => address) public registeredAssets;
28      // values used by system, e.g., min (or max) fee required in game/market
29      mapping(bytes32 => uint256) public globalParameters;
30      // game paused state statuses
31      mapping(address => bool) public activeGames;
```

Contract defines a mapping as registeredAssets, which has not been used throughout the audit scope.

Recommendation

Consider removing unused mapping in order to reduce the contract size and increase code readability.

Status

Fixed

Removed e9da49b6dda146f29681e4b23d9c9c06c6abafaa

C. ERC1155DynamicURI

High Severity Issues

No issues found

Medium Severity Issues

C1. Mutable URIs

```
52      function setURI(uint256 id, string memory newUri)
53          public
54          onlyAuthorized(MINTER_ROLE)
55      {
56          require(_isSetURIRevoked[id] == false, "setter role revoked for id");
57
58          _uris[id] = newUri;
59
60          emit URI(newUri, id);
61      }
62
```

Function allows Authorities to modify URI of any SemiFungible or NonFungible id at any point in time. Authorities with malicious intent may use the function to exploit the characteristics/ traits of a token.

Example Scenario

1- Assume two NFTs X and Y, worth 1ETH and 1000ETH respectively. X has less value because it doesn't have any special/rare character traits. However, Y is a rare NFT.
Authority can change the URI of X to point to the same URI as Y, as a malicious attempt to cheat the system.

2- Authorities can even revoke the privilege of an NFT to ever have a URI. Authorities can call this function for an NFT id, which is not yet minted or the next id to be minted, and thus revoking the privilege for this id to ever have a URI.

Recommendation

Consider reviewing the business and operational logic

Status

Acknowledged

Dev Comment: Can only revoke setting URI after a "ipfs-like" CID is set 94e766f8c911542b17c0104babc636bf056945bc

Low Severity Issues

No issues found

Informational Issues

No issues found

D. Users(Renamed to YoloWallet)

High Severity Issues

No issues found

Medium Severity Issues

D1. Updating Liquidity Pool Balance incorrectly

```
95      function updateLiquidityPoolBalance(uint256 amount)
96          external
97          onlyAuthorized(LIQUIDITY_POOL)
98      {
99          userBalances[msg.sender] += amount;
100     }
106 function reduceLiquidityPoolBalance(address receiver, uint256 amount)
107     external
108     onlyAuthorized(LIQUIDITY_POOL)
109 {
110     userBalances[msg.sender] -= amount;
111
112     yoloTokenContract.transfer(receiver, amount);
113 }
```

Functions allows Authorized roles to update userBalances for Liquidity Pool. However, the function updates the balances for msg.sender and not LP address, which means it won't be possible to manually intervene and update the userBalances for Liquidity Pool in order to maintain the required 1:1 share rate for Yolo and LP shares if needed. Because, if called by any authorized role, the function will be updating userBalances for the authorized roles itself and not for the Liquidity Pool.

Example Scenario

MINTER_ROLE can mint LP shares without increasing the userBalances in Liquidity Pool, thus creating an imbalance with the 1:1 desired share rate, which will require a manual intervention to update userBalances manually in the Users contract. However, as Users contract updates balances for msg.sender and not for LP address, the 1:1 rate will not be fixed until shares are again burned by the MINTER_ROLE.

Recommendation

Consider reviewing and verifying the business and operational logic and replacing msg.sender with Liquidity Pool Address.

Status

Fixed in commit

This was handled in 8a0f004b1416792ff2830d2993c7a85be99b7b8d as requested, but there is still concern that the fix can be bypassed because `reduceLiquidityPool` balance still requires a `receiver` arg.pull request 45

Low Severity Issues

C2. Missing Input Validation

```
124     function gameUpdateUserBalances(
125         address[] memory users,
126         uint256[] memory amounts
127     ) external onlyGameContract {
128         uint256 usersLength = users.length;
129         require(usersLength == amounts.length, "array arg length mismatch");
130 
131         for (uint256 i = 0; i < usersLength; i++) {
132             address user = users[i];
133             uint256 amount = amounts[i];
134             userBalances[user] += amount;
135         }
136     }
144     function gameReduceUserBalance(address user, uint256 amount)
145         external
146         onlyGameContract
147     {
148         userBalances[user] -= amount;
149     }
150 
157     function transferFee(address recipient, uint256 amount)
158         external
159         onlyGameContract
160     {
161         userBalances[recipient] += amount;
162 
163         emit FeeTransfer(recipient, amount);
164     }
```

Functions allow a registered game contract to increase or reduce an user's amount with any value. However, the functions perform no input validation for the user address supplied, which may lead to unexpected results.

Example Scenario

1- A malicious Game Contract can increase/reduce userBalances of its own or another Game Contract
2- A malicious Game Contract can increase/reduce userBalances of Liquidity Pool in order to create an imbalance in the desired 1:1 share rate

Recommendation

Consider reviewing and verifying the business and operational logic and adding required checks in order to reduce the risks of possible scenarios mentioned above.

Status

Acknowledged

Informational Issues

D3. Redundant Check

```
31      constructor(address registryContractAddress_)
32          RegistrySatellite(registryContractAddress_)
33      {
34          require(
35              registryContractAddress_ != address(0),
36              "registry contract address cannot be zero"
37          );
38
39          YoloRegistry registryContract = YoloRegistry(registryContractAddress_);
40
```

constructor adds a Zero Address check for registryContractAddress_ initialization. However, RegistrySatellite already considers the same check,

```
13  abstract contract RegistrySatellite is CoreCommon {
14      YoloRegistry public immutable yoloRegistryContract;
15
16      constructor(address yoloRegistryAddress_) {
17          require(
18              yoloRegistryAddress_ != address(0),
19              "yoloRegistry cannot be zero address"
20          );
21
```

thus making the check, in Users contract redundant, and may be removed.

Recommendation

Consider removing the redundant checks.

Status

Fixed

Redundant checks removed 9c323fb6dec9e94409e0e705b4a6df363dbcd0c0

E. YoloNFTPack

High Severity Issues

E.1 Resetting usersTokens

```
93      function mintBaseSFT() external {
94          // TODO: any hooks or checks
95
96          _mintNFT(msg.sender, BASE_SFT_ID, EMPTY_STR);
97      }
```

Function allows any user to self mint a SemiFungible token of BaseType as BASE_SFT_ID or BaseType 1. However, the function may reset usersTokens,

```
266
267         uint256 id = baseType | index;
268
269         _nftOwners[id] = to;
270         usersTokens[to] = id;
271         totalBaseTypeBalances[baseType]++;
272
273         super._mint(to, id, UNITY, EMPTY_BYTES);
```

if the user holds a token from another BaseType, and may impact a huge loss to the token holder.

Example Scenario

Assume a user holds an expensive NFT or upgraded its Token to a higher level. Calling this function will reset the usersTokens to BaseSFT, thus impacting a huge loss

Recommendation

Consider adding necessary checks in order to avoid risks of possible scenarios mentioned above.

Status

Fixed

Comment: Team Added an OnlyAuthorized role for the same

Medium Severity Issues

No issues found

Low Severity Issues

E2. Spam Minting

```
93      function mintBaseSFT() external {
94          // TODO: any hooks or checks
95
96          _mintNFT(msg.sender, BASE_SFT_ID, EMPTY_STR);
97      }
```

Function allows any user to self mint a SemiFungible token of BaseType as BASE_SFT_ID or BaseType 1. However, no access control may lead to spam minting.

Example Scenario

- Self-Mint a BaseSFT
- Burn it down
- Self-Mint another one

Thus, by repeating the process a user may utilize the token Index space of the BaseType and may disallow other users to mint more BaseSFTs

Recommendation

Consider adding access control or necessary checks in order to avoid spam minting.

Status

Fixed

Informational Issues

No issues found

F. NFTTracker

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

F1. Unsafe DownCasting and “uint” Space Mismatch

```
156         uint256 tokenBase = tokenIndex.getBaseType();
157
158         LevelTracking storage levelTracker = levelTrackingMap[tokenBase];
159
160         nftTracking.cumulativeBidAmount += amount;
161         levelTracker.totalCumulativeBidAmount += uint160(amount);
162
```

Function updateTracking allows the registered game contracts to update NFT/LEVEL tracking details. However, at #L161, the function performs an unsafe downcasting from uint192 to 160, which may lead to incorrect details being updated. Also, the
NftData holds a type uint192 for cumulativeBidAmount, whereas LevelTracking holds a type uint176 for totalCumulativeBidAmount

```
18      struct NftData {
19          uint64 bidCount;
20          uint192 cumulativeBidAmount;
21          mapping(bytes32 => mapping(uint256 => bool)) hasUserBid;
22      }
23
24      struct LevelRequirement {
25          uint64 bidCountThreshold;
26          uint192 cumulativeAmountThreshold;
27          uint256 nextLevelId;
28          uint256 prevLevelId;
29      }
30
31      struct LevelTracking {
32          uint64 totalBidCount;
33          uint176 totalCumulativeBidAmount;
34      }
```

Which means, cumulativeBidAmount of an NFT can even exceed the totalCumulativeBidAmount of the level, which seems logically incorrect.

Recommendation

Consider reviewing the business and operational logic.

Status

Fixed

Comment: The team removed redundant require checks at commit d4d921718ecb24ef4d5567bbb1751b386df18da9

Informational Issues

F2. Logical Code Optimization#1

```
205         require(bidCountThreshold > 0, "bid threshold must be g.t. 0");
206         require(
207             cumulativeAmountThreshold > 0,
208             "amount threshold must be g.t. 0"
209         );
```

While setting Level Requirement, there is no need for the checks mentioned above, as for the first level, there will be no previous level(can be considered level 0), meaning prevLevel.bidCountThreshold and prevLevel.cumulativeAmountThreshold will always be 0. So the following check

```
234         require(
235             bidCountThreshold > prevLevel.bidCountThreshold &&
236                 cumulativeAmountThreshold > prevLevel.cumulativeAmountThreshold,
237             "new thresholds must be greater than lower level"
238         );
239
```

Will be taking care of the values to be greater than 0. So, the first level can have a minimum value 1 for the bidCountThreshold and cumulativeAmountThreshold. Also, as the check makes sure that the new level's bidCountThreshold and cumulativeAmountThreshold stay greater than the previous level, the following levels will always have non-zero value for bidCountThreshold and cumulativeAmountThreshold.

Recommendation

The code can be refactored based on the optimization suggested above.

Status

Fixed

Removed redundant require checks d4d921718ecb24ef4d5567bbb1751b386df18da9

F3. Logical Code Optimization#2

```
80      function getNFTLevelsListRange(uint256 startIndex, uint256 length)
81          public
82          view
83          returns (uint256[] memory nftLevels)
84      {
85          // TODO: consider design to return min(length, _nftLevelIds.length - startIndex)
86
87          require(
88              startIndex + length <= _nftLevelIds.length,
89              "range out of array bounds"
90          );
91
```

Function can be optimized to add a non-zero value check for parameter length, as it doesn't make any sense to return 0 elements from any starting index.

Recommendation

The code can be refactored based on the optimization suggested above.

Status

Fixed

Fixed in: 7e89e1c628713a07b18e6614ecdb6b48d1a804ce

G. GameInstance

High Severity Issues

G1. Missing sanity value checks on amount in makeMarketBid

It is possible to bid 0 amount or even very low amounts via the makeMarketBid function. Thus the MAX_BIDS limit can be reached quickly if one is allowed to bid 0 amount of tokens. This could result in manipulation of gamerounds in order to win or deny other people from bidding.

```
174     function makeMarketBid(uint256 bidRound↑, uint256[2] calldata amounts↑)
175         external
176         override
177         onlyAuthorized(MARKET_MAKER_ROLE)
178         whenNotPaused
179     {
180         require(bidRound↑ > roundIndex, "cannot bid in live round");
181         require(
182             bidRound↑ <= 10 + roundIndex,
183             "cannot bid more than 10 rounds in advance"
184         );
185         require(amounts↑[0] + amounts↑[1] < marketLimit, "amount exceeds limit");
186 
187         // first bid up amount
188         _bidInYolo(amounts↑[0], true, bidRound↑, lpAddress);
189         // then bid down amount
190         _bidInYolo(amounts↑[1], false, bidRound↑, lpAddress);
191     }
192 
```

Example Scenario

It is possible to pass amounts[0] = 0, amounts[1] = marketLimit - 1

Recommendation

Consider adding appropriate require checks for the same.

Status

Acknowledged

Comment: The team said that it was not rational for them to attack their own product, as in version 1, and they will have only access to this function.

Medium Severity Issues

G2. Centralization of processRound

processRound() function is used to make the round live so that it cannot accept furrther bids and give winning players their payout. But this can only be called by GAME_ADMIN. This could resulting in the admin denying the players their payouts for a long time and thus resulting in Denial of Service.

```
222     function processRound(
223         uint256 startTime↑,
224         uint256 settlementPrice↑,
225         uint256 nextStrikePrice↑
226     ) external override onlyAuthorized(GAME_ADMIN_ROLE) {
227         require(
228             settlementPrice↑ > 0 && nextStrikePrice↑ > 0,
229             "args must be g.t. 0"
230         );
231
232         // is this constraint required?
233         require(
234             startTime↑ >= roundDatas[roundIndex].startTime + GAME_LENGTH,
235             "min duration for start required"
236         );
237
238         _calculatePayouts(settlementPrice↑);
239
240         roundIndex++;
241
242         _startRound(startTime↑, nextStrikePrice↑);
243     }
244
```

Recommendation

It is advised to make this function more decentralized so that there is less reliance on GAME_ADMIN to make a payout to winning players as the GAME_ADMIN can indefinitely delay the payout of winning players.

Status

Fixed

G3. Missing value checks on startTime parameter

startTime parameter of processRound() function has no upper limit and can be set incorrectly as very long time in future. This could result in a potential Denial of Service.

```
222     function processRound(
223         uint256 startTime,
224         uint256 settlementPrice,
225         uint256 nextStrikePrice
226     ) external override onlyAuthorized(GAME_ADMIN_ROLE) {
227         require(
228             settlementPrice > 0 && nextStrikePrice > 0,
229             "args must be g.t. 0"
230         );
231
232         // is this constraint required?
233         require(
234             startTime >= roundDatas[roundIndex].startTime + GAME_LENGTH,
235             "min duration for start required"
236         );
237
238         _calculatePayouts(settlementPrice);
239
240         roundIndex++;
241
242         _startRound(startTime, nextStrikePrice);
243     }
244
```

Example Scenario

It is possible to set the startTime parameter as 1 year in future.

Recommendation

It is advised to add appropriate require checks for the same such as an upper limit check on startTime parameter.

Status

Fixed

Dev Comments: The team added MAX_START_DELAY constraint for startTime constraint at commit 898df4d4a82d40dccaa109642eff41d4c34c6074

G4. Possible manipulation of settlementPrice and nextStrikePrice

The settlementPrice and nextStrikePrice parameters can be manipulated by admin in order to favour specific bidders.

```
222     function processRound(
223         uint256 startTime↑,
224         uint256 settlementPrice↑,
225         uint256 nextStrikePrice↑
226     ) external override onlyAuthorized(GAME_ADMIN_ROLE) {
227         require(
228             settlementPrice↑ > 0 && nextStrikePrice↑ > 0,
229             "args must be g.t. 0"
230         );
231
232         // is this constraint required?
233         require(
234             startTime↑ >= roundDatas[roundIndex].startTime + GAME_LENGTH,
235             "min duration for start required"
236         );
237
238         _calculatePayouts(settlementPrice↑);
239
240         roundIndex++;
241
242         _startRound(startTime↑, nextStrikePrice↑);
243     }
244
```

Example Scenario

If 10 bidders bid for up, 5 bidder bid for down and the admin wants down to win in order to favour a specific bidder, then even if in reality Up won, the admin can set the the strikeprice and settlementPrice such that Down wins instead.

Recommendation

It is advised to use Decentralized Oracles for fetching and then calculating the settlementPrice and the nextStrikePrice.

Status

Acknowledged

Dev Comments: The team acknowledged to add decentralized oracles in future in order to calculate the same

Low Severity Issues

G5. Missing zero value checks for marketLimit

```
162     * @notice Grab market limit periodically from {LiquidityPool} to save on external call costs
163     * @dev This is called in sync with market limit changes in {LiquidityPool}.
164     **/
        ftrace | funcSig
165     function acquireMarketLimit() external onlyAuthorized(ADMIN_ROLE) {
166         marketLimit = LiquidityPool(lpAddress).marketLimit();
167     }
168
```

There is no zero value check for the marketLimit. If the marketLimit is not set in the LiquidityPool contract or it is not updated in this contract(that is it is zero), it will result in makeMarketBid() function always failing due to the require statement on line: 185.

Recommendation

Add require checks for the same.

Status

Fixed

Client's Comment: Intentional design to limit market bidding until activated or in case it needs to be deactivated

Informational Issues

No issues found

H. GameInstanceWithNFTPack

High Severity Issues

H1. Users can harvest more rewards than expected

updatetracking is done in bidInYolo based on the nft id for tracking bids for cumulative rewards which is fetched from usersTokens. But usersTokens from yoloNFTPack (which inherits from ERC1155SemiFungible) is used to fetch the token id which could get overwritten each time a token is transferred to an account.

A user can exploit this by minting new nft for himself, then placing a bid such that bidCount increases by 1, then claiming rewards from BiddersRewards by calling harvest, then again minting a new nft that overwrites its previous nft tracked by usersTokens, then placing a bid such that bidCount increases by 1(possible because the id of this new nft is different) and then claiming rewards by calling harvest again.

harvest() function gives more weight to the bidCount. So this exploit can be highly profitable to any user.

```
73      function bidInYolo(
74          uint256 amount↑,
75          bool isUp↑,
76          uint256 bidRound↑
77      ) public override {
78          super.bidInYolo(amount↑, isUp↑, bidRound↑);
79
80          uint256 tokenId = yoloNFTPackContract.usersTokens(msg.sender);
81
82          // TODO: decide if this a requirement
83          if (tokenId > 0) {
84              nftTrackerContract.updateTracking(
85                  tokenId,
86                  uint192(amount↑),
87                  GAME_ID,
88                  bidRound↑
89              );
90          }
91      }
92  }
93
```

Recommendation

It is advised that usersToken is not overwritten and to disallow users from minting NFT again and again. The team can also consider into directly tracking the address of the bidder and adding sufficient require checks in order to prevent such scenarios from happening.

Status

Acknowledged

Client's Comment: The SFTs can only be minted after a user is whitelisted (and currently pays 10USD to do so). This should remove the attack vector. Also, any user with a balance cannot receive transfers of token.

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

Informational Issues

No issues found

I. LiquidityPool

High Severity Issues

I1. Incorrect Implementation: The Circus

```
95      function mintInitialShares(uint256 initialAmount)
96          external
97          whenNotLPBalance
98      {
99          require(totalSupply() == 0, "Initial share amount already created");
100
101         address sender = msg.sender;
102
103         yoloTokenContract.transferFrom(
104             sender,
105             address(usersContract),
106             initialAmount
107         );
108
109         _mint(sender, initialAmount);
110
111         usersContract.updateLiquidityPoolBalance(initialAmount);
112
113         hasLPTokensCirculating = true;
114     }
```

Function allows anyone to mint initial shares to the system. However, there is no check for the initialAmount, and anyone can call this function with a 0 value for the initialAmount. Minting 0 initial shares will switch the boolean hasLPTokensCirculating to true, even though there is logically no LP Supply in circulation. As the boolean value has now turned to true, it means the contract will now allow to mint more LP shares.

```
121     function mintLpShares(uint256 depositAmount) external whenLPBalance {
122         address sender = msg.sender;
123
124         yoloTokenContract.transferFrom(
125             sender,
126             address(usersContract),
127             depositAmount
128         );
129
130         // should be 1:1 with current implementation
131         uint256 newShareAmount = (totalSupply() * depositAmount) /
132             usersContract.userBalances(address(this));
133
```

But since we minted 0 shares and didn't transferred any Yolo Tokens to users contract and din't update the userBalances at users contract. The userBalances still holds 0.
Now trying to call mintLpShares, will revert due to divide by 0 panic.
So, how should we tackle the scenario now?
Can we increase the userBalances manually from users contract?
No, referring to D.1. The updateLiquidityPoolBalance updates userBalances of msg.sender and not LP address, which means any attempt to call this function will be increasing the balance of the caller itself.
Can we call mintInitialShares again with the correct amount?
No, as the modifier whenNotLPBalance, will not allow us to do that
Can we switch the boolean hasLPTokensCirculating to false?
Can be done by calling burnLpShares, but since there is no LP supply,

```
144     function burnLpShares(uint256 burnAmount) external {
145         address sender = msg.sender;
146         // !!! must call supply before burn
147         uint256 sharesTotalSupply = totalSupply();
148
149         _burn(sender, burnAmount);
150
151         uint256 tokenTransferAmount = (burnAmount *
152             usersContract.userBalances(address(this))) / sharesTotalSupply;
153
```

It will again result into a divide by 0 panic.
What can be done?
The possible way is, MINTER_ROLE can mint 1 token in order to avoid this divide by 0 panic, and then call burnLpShares to switch hasLPTokensCirculating back to false.
The above Circus indicates that the logical implementation of modifiers whenNotLPBalance and whenLPBalance is incorrect.

Exploit Scenarios

- Call mintInitialShares with 0 initialAmount, in order to mess up the functions
- Mint some initial shares and later burn them down(ERC20 burn function which will not switch hasLPTokensCirculating back to false). Indeed a griefing attack, but invalidates the logical reasoning of the contract, that the contract is having LP supply because the functions rely on a boolean value rather than the actual supply.

Recommendation

There is no need to check a boolean value, in order to allow/disallow the calls to mintInitialShares and mintLpShares, which we have already seen with the above scenarios, that they are logically incorrect. The simplest way is to check the totalSupply itself, i.e allow minting initial shares only if there is no totalSupply which logically makes sense as there is no LP Supply in circulation and which is already being checked,

```
95      function mintInitialShares(uint256 initialAmount)
96          external
97          whenNotLPBalance
98      {
99          require(totalSupply() == 0, "Initial share amount already created");
100
101         address sender = msg.sender;
102
```

and if there exists any LP shares, then only allows calls to mintLpShares. Similarly, the totalSupply can be checked for the calls to burnLpShares that is to allow burning, only if there exists any LP Supply, and there is no need to rely on a boolean value and reset it.

```
158         if (sharesTotalSupply - burnAmount == 0) {
159             hasLPTokensCirculating = false;
160         }
```

Status

Fixed

I2. Exploiting the desired 1:1 Yolo:LPShare rate & Possible RugPulls

```
121     function mintLpShares(uint256 depositAmount) external whenLPBalance {
122         address sender = msg.sender;
123
124         yoloTokenContract.transferFrom(
125             sender,
126             address(usersContract),
127             depositAmount
128         );
129
130         // should be 1:1 with current implementation
131         uint256 newShareAmount = (totalSupply() * depositAmount) /
132             usersContract.userBalances(address(this));
133
134         _mint(sender, newShareAmount);
135
136         usersContract.updateLiquidityPoolBalance(depositAmount);
137     }
```

MINTER_ROLE can mint any number of tokens in order to increase the totalSupply and disrupt the 1:1 share rate.

```
62      function mint(address to, uint256 amount) public virtual {
63          require(
64              hasRole(MINTER_ROLE, msg.sender),
65              "ERC20PresetMinterPauser: must have minter role to mint"
66          );
67          _mint(to, amount);
68      }
```

As the mint function doesn't account for the Yolo Tokens that are supposed to be transferred into users contract and also doesn't update the userBalances in the users contract. It can lead to multiple exploit scenarios.

Exploit Scenarios

- MINTER_ROLE can mint some shares for itself, without even sending any yolo tokens to the users contract, in order to extract yolo tokens deposited by other users
- MINTER_ROLE can mint any number of shares to any random account, in order to disrupt 1:1 share rate, and as the totalSupply has been increased, the new deposits will earn more shares, even though there is no yolo liquidity in the users contract
- Any user can burn its shares down. Indeed a griefing attack, but leads to disrupting the desired 1:1 share rate.

Recommendation

Consider reviewing and verifying the business and operational logic

Status

Fixed

I3. Use of SafeTransfer for Third party Token Contract

Description

Use safeTransferFrom for token transfers as USDC is an upgradeable contract, or check return values of token transfers. Needs to be fixed for every instance in every contract.

Remediation

We recommend using safeTransfer, until and unless you are 100% sure about the returned values of the token transfers. But we think the token would be the same throughout the lifespan of contract

Status

Fixed in *pull request 75*

I4. Decimal Error

Description

USDC has 6 decimals and LPool has 18. Means while minting they are not going to get exact amount. For instance, Trying to mint 5 USDC worth of shares will provide only 5e-12 shares, thus creating visualization issues for the users.

Remediation

If Yoloteam are implementing a change, then there is a need to add multiplying factors accordingly, to balance the logic. kindly check the instances of decimals, wherever you made the changes (from 18 decimal yolo to 6 decimal USDC)

Status

Fixed in *pull request 66*

Medium Severity Issues

I5. StableCoinToken is supposed to be an interface

ERC20 public immutable stablecoinTokenContract;
YoloWallet public immutable walletContract;

Status

Fixed in *pull request 66*

Fixed in: cd3de5aca7f6133d6c5b3ca0a3ee09eb73311b24
Client's Comment: This was in order to access a method in ERC20, but no longer needed.

I6. ERC20Burnable comes with one more burn function that burnFrom

```
function burnFrom(address account, uint256 amount) public virtual {
    _spendAllowance(account, _msgSender(), amount);
    _burn(account, amount);
}
```

yolosharetokens contract comes with burnFrom function as well, which imposes the same risk of unintended share manipulation. If devs plan to revert this function as well, then ERC20Burnable can be removed from yolosharetokens as there will be no more use afterwards.

Status

Acknowledged

I7. The logical requirement is that liquidity...

The logical requirement is that liquidity providers should maintain a minimum amount of 400 USDC. However, they can bypass it by burning down the shares, which they don't require, thus tricking the system and avoding the minimum requirement.

Remediation

Consider verifying the business and operational logic

Status

Fixed

Low Severity Issues

I8. [L#104]setProtectionFactor missing lower bound/ upper bound value checks

```
/
*
*
    * @notice Sets `protectionFactor` value as part of additional guard layer on higher frequency
  `marketLimit` adjustments. See: `setMarketLimit` below.
    * @dev This value should float between ~500-20000 and updated only on big pool swings.
    * @param newFactor Simple factor to denominate acceptable marketLimit value in `setMarketLimit`.
    **/
    function setProtectionFactor(uint256 newFactor)
      external
      onlyAuthorized(ADMIN_ROLE)
    {
      protectionFactor = newFactor;
    }
```

Commit hash: 5ce6a9c7e0aa6908e634fe939c4159e88235f249

The dev comment says protection factor should float from ~500-20000, but setProtectionFactor doesn't contain any lower bound / upper bound value checks

Status

Acknowledged

Dev Comment: Can be corrected at any time with a trivial call from admin.

Informational Issues

I9. Unused MARKET_MAKER_ROLE

```
71          );
72
73          yoloTokenContract = YoloEthereumUtilityTokens(yoloUtilityTokenAddress);
74
75          usersContract = Users(userBalancesAddress);
76
77          _grantRole(MARKET_MAKER_ROLE, msg.sender);
78      }
79
```

The contract initialization grants MARKET_MAKER_ROLE to the deployer. However, it is not being used for any Access Control, for the scope of the contract.

Recommendation

Consider reviewing and verifying the business and operational logic

Status

Fixed

I10. burnLPShares will result into divide by zero panic, if there is no totalSupply

```
/**
     * @notice Burns LP shares in exchange for share of pool USDC tokens.
     * @dev  Will require share token approval from sender to contract to burn.
     * @param burnAmount Amount of LP share to burn for USDC withdrawal.
     **/
    function burnLpShares(uint256 burnAmount) external {
    address sender = msg.sender;
    // !!! must call supply before burn
    uint256 sharesTotalSupply = totalSupply();
```

Status

Acknowledged

J. ERC1155SemiFungible

High Severity Issues

J1. Exploiting usersTokens

```
264         // increment maxIndexes first, THEN assign index
265         uint256 index = ++maxIndexes[baseType];
266
267         uint256 id = baseType | index;
268
269         _nftOwners[id] = to;
270         usersTokens[to] = id;
271         totalBaseTypeBalances[baseType]++;
272
273         super._mint(to, id, UNITY, EMPTY_BYTES);
```

The usersTokens holds the token ID of a user. However, it can only hold the most recent token ID, the user has acquired, thus opening doors for overwriting an existing token ID, if the user acquires any new token, which may impact a user in losing a valuable token.

Example Scenario

1- Authorized roles can mint new token IDs in order to overwrite any user's existing token ID

Let's assume, a user is having a token from basetype1, the authorized roles can mint a token of basetype2, thus overwriting the existing token ID.

2- A user may lose its existing token by itself.

Referring E.1 mintBaseSFT function allows anyone to self-mint a basetype1 semi fungible token. Let's assume a user already holds a semi-fungible token from another basetype or a non-fungible token. By minting the base SFT, the user will lose its existing token.

3- Exploiting Upgraded tokens

Let's assume, a user has upgraded its token, let's assume to a level 3. Authorized roles can mint a lower level token, let's say a level 1 token, impacting the users a huge loss.

Recommendation

Consider reviewing and verifying the business and operational logic.

Status

Fixed

J2. Resetting values incorrectly while burning a token ID

```
225     function burn(
226         address account,
227         uint256 id,
228         uint256
229     ) public override {
230         usersTokens[msg.sender] = 0;
231         _nftOwners[id] = address(0);
232
233         uint256 baseType = id.getBaseType();
234
235         _hasNFTType[baseType][msg.sender] = false;
236
237         super.burn(account, id, UNITY);
238
239         totalBaseTypeBalances[baseType]--;
240     }
```

Function burn allows a user to burn its token ID. However, the function doesn't account for the scenarios where the operator can call this function on behalf of the token ID holder, as a consequence, the operator may reset its own usersTokens and _hasNFTType, instead of the concerned account.

Recommendation

The function should reset usersTokens and _hasNFTType of the account and not msg.sender, considering the scenarios where an operator can burn the token ID on behalf of the account.

Status

Fixed

Medium Severity Issues

J3. Tricking the system to hold more than 1 token from same basetype

```
162     function safeTransferFrom(
163         address from,
164         address to,
165         uint256 id,
166         uint256,
167         bytes calldata
168     ) public override {
169         if (id.isSemiFungible() || id.isNonFungible()) {
170             // require(balanceOf(from, id) == UNITY, "from address must be owner");
171             // require(balanceOf(to, id) == 0, "already (n|s)ft series owner");
172
173             // TODO: kludge to prevent token overwrite attacks until we decide on multitoken holdings support
174             require(usersTokens[to] == 0, "receiver already has a token");
175
176             _nftOwners[id] = to;
177             usersTokens[from] = 0;
178             usersTokens[to] = id;
179
180             uint256 baseType = id.getBaseType();
181
182             mapping(address => bool) storage hasBase = _hasNFTType[baseType];
183
184             hasBase[from] = false;
185             hasBase[to] = true;
186         }
```

The logical requirement of the team/yolo ecosystem to not allow users to have more than 1 token from same basetype. But that can be tricked with the help of safeTransferFrom and safeBatchTransferFrom function.

Example Scenario

Let's assume, a user is having two tokens from basetype1 and basetype2, and let's assume it wants to have one more token of basetype1. It can send basetype2 token to another account, thus resetting usersTokens to 0, and allowing it to receive a token of any basetype(here basetype1) because the function doesn't check for the receiver's basetype, if it holds any token.
Now, it can receive a token of basetype1, and as a result, will be holding two tokens of the same basetype.

Recommendation

Consider adding checks for the receiver's basetype, if it holds any token

Status

Fixed

Low Severity Issues

J4. Non-Executable Code

```
280     function mint(
281         address,
282         uint256,
283         uint256,
284         bytes memory
285     ) public pure override {
286         revert("mint disabled");
287     }
288
289     // remove mintBatch functionality as it circumvents design restrictions of this contract
290     function mintBatch(
291         address,
292         uint256[] memory,
293         uint256[] memory,
294         bytes memory
295     ) public pure override {
296         revert("mintBatch disabled");
297     }
298
299     function burnBatch(
300         address,
301         uint256[] memory,
302         uint256[] memory
303     ) public pure override {
304         revert("burnBatch disabled");
305     }
```

Multiple instances of non-executable code have been reported, which may be removed to increase the code readability and decrease code size.

Recommendation

Consider reviewing and verifying the business and operational logic and consider removing the non-executable code.

Status

Fixed

J5. Insufficient uint space to hold possible token indexes/IDs

```
39      // TODO: discuss necessity of type existence checking, validation vs efficiency tradeoff
40      mapping(uint256 => bool) public typeBirthCertificates;
41      // gets token id for provided address; inverse of _nftOwners
42      mapping(address => uint256) public usersTokens;
43      // gets number of SFT/NFTs belonging to base type
44      mapping(uint256 => uint120) public maxIndexes;
45      // total balance by nft level
46      mapping(uint256 => uint256) public totalBaseTypeBalances;
47
48      constructor() ERC1155PresetMinterPauser(EMPTY_STR) {}
49
```

The maxIndexes act as a nonce to mint new token Indexes/IDs. The maximum possible value that it can hold or in other words the maximum number that can be minted as token ID is 2^120-1. However, the number of possible token IDs that can be minted for a basetype are 2^128-1(starting from 1), which means, it is not possible to mint all the possible token IDs

Recommendation

Consider reviewing and verifying the business and operational logic and choosing an appropriate uint space.

Status

Acknowledged

Informational Issues

J6. Insufficient uint space to hold possible baseTypes

```
33      uint120 private _nftNonce;
34      uint120 private _semiFtNonce;
35
```

Referring to A.1, _nftNonce and _semiFtNonce are supposed to act as counter/nonce to create new baseTypes. They are of type uint120 and the maximum value they can hold is 2^120-1. However, the possible base types that can be created are 2^126-1 (considering 1 as the first baseType), as a consequence, the nonces will not able to accommodate all the possible base types.
Contrary, it acts as a protection to avoid collisions(Refer. A.1) of base types that can happen after value 2^126-1, as the next uint type after uint120 is uint128, now opting to uint128 will open doors for collisions as the values can go more than 2^126-1

Status

Acknowledged

J7. Unused BaseType 0

```
111     function createNFTBaseType(bool _isSFT)
112         external
113         onlyAuthorized(MINTER_ROLE)
114     {
115         uint256 baseType;
116         // Store the type in the upper 128 bits
117         if (_isSFT) {
118             baseType = (uint256(++_semiFtNonce)).encodeNewSemiFungibleBaseType();
119             // console.log("Binary %s", (baseType).u256ToBinaryStr());
120         } else {
121             baseType = (uint256(++_nftNonce)).encodeNewNonFungibleBaseType();
122         }
123
```

As the basetypes are created by incrementing _nftNonce and _semiFtNonce down. The first basetype will start from 1 and basetype 0 for NFT and SemiFT which are TYPE_NF_BIT and TYPE_SEMI_BIT themselves, will remain unused.

Status

Acknowledged

K. BiddersRewards

High Severity Issues

K1. Exploiting rewards#1

```
143     function removeBiddersRewardsContract()
144         external
145         onlyAuthorized(ADMIN_ROLE)
146     {
147         biddersRewardsContract = BiddersRewards(address(0));
148
149         emit AddressSet(BIDDERS_REWARDS, address(0));
150     }
```

NFTTracker allows authorized roles to remove biddersRewardsContract at any point of time, as a consequence NFTTracker will not update tracking for any levels/NFTs thereafter in biddersRewardsContract

```
212         if (address(biddersRewardsContract) != address(0)) {
213             biddersRewardsContract.updateTracking(
214                 tokenIndex,
215                 newRoundBid,
216                 amount
217             );
218         }
219
```

As a result, the participation units for the NFT will not be updated and for the same reason, the level weightings will not be updated, thus disallowing the user to harvest the intended rewards.

Exploit Scenario

Let's assume, there exists 2 NFT Ids as N1 & N2 in level 1.
N1 with bidCount as 1 and cumulativeBidAmount as 250.
N2 with bidCount as 1 and cumulativeBidAmount as 250.
Now in order to exploit, the authorized role removes the biddersRewardsContract from NFTTracker.
Now, let's say a user with id N1 placed another bid, with an amount of 250. It would have increased the participation units for this id, which would have generated more rewards for this Id, but since there is no more biddersRewardsContract in NFTTracker, the new tracking details will not be updated in the bidders rewards, thus exploiting rewards for this id.

Recommendation

Consider reviewing and verifying the operational and business logic.

Status

Acknowledged

Dev Comment: Rewards are still tracked in the master tracking account i.e. NFTTracker and can be referenced by future contracts where needed. Without this if condition, the current tracker can break and disrupt intended execution.

K2. Exploiting rewards#2

```
394     function updateTracking(
395         uint256 tokenIndex,
396         uint256 newRoundBid,
397         uint192 amount
398     ) external onlyAuthorized(NFT_TRACKER) {
399         NftData storage nftTracking = epochTokenTracker[tokenIndex];
400
```

```
417     function bumpDuringUpgrade(uint256 oldTokenId, uint256 newTokenId)
418         external
419         onlyAuthorized(YOLO_NFT_PACK)
420     {
421         NftData storage oldNftTracking = epochTokenTracker[oldTokenId];
```

Functions updateTracking and bumpDuringUpgrade are supposed to be called by NFTTracker and YoloNFTPack respectively. However, the modifier utilized i.e onlyAuthorized expands this capability to anyone having:

- NFT_TRACKER or YOLO_NFT_PACK role in BiddersRewards
- NFT_TRACKER or YOLO_NFT_PACK role in associated YoloRegistry of BiddersRewards
- DEFAULT_ADMIN_ROLE or ADMIN_ROLE role in associated YoloRegistry of BiddersRewards

Exploit Scenarios

Let's assume, there exists 2 NFT Ids as N1 & N2 in level 1.

N1 with bidCount as 1 and cumulativeBidAmount as 250.

N2 with bidCount as 1 and cumulativeBidAmount as 250.

1 - Exploiting Participation Units

Now in order to exploit rewards of N1, the authorized role may increase the bidCount and cumulativeBidAmount for N2, thus increasing its participation units and allowing it to harvest more rewards.

2 - Exploiting Level Weightings

Authorized roles may increase the bidCount and cumulativeBidAmount for any existing ID of any level, prior to releasing funds. Increasing bidCount and cumulativeBidAmount will increase totalBidCount and totalCumulativeBidAmount too of the level, thus increasing its level weighting. Authorized roles may do this to generate more rewards for an existing level, and thereafter extract them for a specific ID(as they can manipulate participation units too).

3 - bumpDuringUpgrade

It may be manually called by any authorized role to transfer participation units from any existing ID to another ID and manipulate level weightings anytime, in order to exploit intended rewards.

Recommendation

Consider reviewing and verifying the operational and business logic. The calls may be restricted to NFTTracker and YoloNFTPack only for the concerned functions, by modifying the modifier's logic, in order to reduce the attack surface.

Status

Acknowledged

K3. Dividing Rewards and Double Harvesting

As we know releasing funds is a one-time operation that calculates level weightings and intended rewards. However, there exists no check to restrict biddings once the funds are released

```
306     function releaseFunds() external {
307         require(
308             isReadyToProcess || block.timestamp > startTime + 30 days,
309             "funds must be released or seasoned 30 days"
310         );
311
312         // get nft levels list and grab highest level for the highest multiplier and THEN require best reward per block lower than amount sent
313
314         uint256 nftLevelIdsListLength = nftTrackerContract
315             .getNFTLevelIdsLength();
316         uint256[] memory levelWeightings = new uint256[](nftLevelIdsListLength);
317         uint256[] memory nftLevelIdsList;
318         uint256 weightingSum;
```

As a result, newer bids may extract rewards that were intended for other ids.

Example Scenario

1- Let's assume, at the time of releasing funds, there existed only 1 NFT Id as N1 in level 1, which means all the rewards for this level, is supposed to be distributed to id N1. However, let's assume another user got an id N2 of the same level. It can now bid in order to have participation units. Now the same reward will be divided into two IDs N1 and N2, which was intended only for N1

2- Double Harvesting: Let's assume, id N1 is a genuine participant and should be getting the intended rewards. After harvesting rewards, the user may upgrade its token to get a token, let's say N2 of the next level. Upgrading a token, transfers participation units from old id to new id, which means, the user will be eligible to harvest rewards for the new id as well, which it just got from upgrading.

Recommendation

Consider reviewing and verifying the operational and business logic.

Status

Fixed

Dev Comments: The BiddersRewards contracts will be rotated via BiddersRewardsFactory, in order to disallow bids to update tracking once the funds are released.

Medium Severity Issues

K4. Incorrect Implemented Logic to calculate Level Rewards

```
162     function getLatestLevelReward(uint256 id)
163         public
164         view
165         returns (uint256 levelYoloReward)
166     {
167         uint256 baseType = id.getBaseType();
168
169         uint256 yoloReward = yoloTokenContract.balanceOf(address(this));
170         uint256 totalLevelWeighting = getTotalLevelWeighting(baseType);
171         uint256 allLevelsWeighting = getCombinedLevelsWeighting();
172
173         levelYoloReward =
174             (yoloReward * totalLevelWeighting) /
175             allLevelsWeighting;
176     }
177
```

The intended logic of the function is to allow users to view expected level rewards at any point in time. However, the function logic doesn't consider multiplying rewardsMultiplier for the concerning level and thus will produce incorrect results.

For the same reason, getUserPendingReward will produce incorrect results as it depends upon getLatestLevelReward to calculate the level's reward.

```
130     function getUserPendingReward(uint256 id)
131         external
132         view
133         returns (uint256 pendingReward)
134     {
135         uint256 participationWeight = getUserParticipationWeight(id);
136
137         if (!harvestLogs[id]) {
138             uint256 totalLevelWeighting = getTotalLevelWeighting(id);
139             uint256 latestYOLOInLevel = getLatestLevelReward(id);
140
141             pendingReward =
142                 (participationWeight * latestYOLOInLevel) /
143                 totalLevelWeighting;
144         } else {
145             pendingReward = 0;
146         }
147     }
```

Recommendation

Consider reviewing and verifying the operational and business logic and multiplying rewardsMultiplier of baseType with calculated totalLevelWeighting

Status

Fixed

Low Severity Issues

K5. DoS with rewardsMultiplier

```
329         for (uint256 i = 0; i < nftLevelIdsListLength; i++) {
330             uint256 nftLevelId;
331             uint256 rewardsMultiplier;
332
333             nftLevelId = nftLevelIdsList[i];
334             rewardsMultiplier = nftTrackerContract.rewardsMultipliers(
335                 nftLevelId
336             );
337
338             require(rewardsMultiplier > 0, "set all rewards multipliers");
339
```

Referring to L3, releaseFunds has a strict condition, that rewardsMultiplier should be set for all the levels, and if any level's rewardsMultiplier is not set, the entire function will revert.

Authorized roles may take advantage and set any existing level's rewardsMultiplier as 0, thus disallowing the release of funds

```
314     function setUserIncentives(uint256 baseIndex, uint16 multiplier)
315         external
316         onlyAuthorized(MINTER_ROLE)
317     {
318         require(
319             baseIndex.isSemiFungibleBaseType() ||
320                 baseIndex.isNonFungibleBaseType(),
321             "incorrect token base encoding"
322         );
323
324         rewardsMultipliers[baseIndex] = multiplier;
325
326         emit UserIncentivesSet(baseIndex, multiplier);
327     }
```

Recommendation

Consider reviewing and verifying the operational and business logic. A check may be added to make sure the multiplier being set is a non-zero value(if the business logic allows).

Status

Acknowledged

K6. Missing checks may lead to incorrect contract initialization

```
105     ) RegistrySatellite(registryContractAddress_) {
106         YoloRegistry registryContract = YoloRegistry(registryContractAddress_);
107
108         address yoloTokenContractAddress = registryContract.getContractAddress(
109             YOLO_TOKEN
110         );
111
112         require(
113             yoloTokenContractAddress != address(0),
114             "token address cannot be zero"
115         );
116
117         yoloTokenContract = IERC20(yoloTokenContractAddress);
118         nftTrackerContract = trackerInstance_;
119         yoloNFTPackContract = nftPackInstance_;
120
```

A different pattern has been observed in contrast with other contracts which refer to YoloRegistry for registered contract addresses. However, here, the nftTrackerContract and yoloNFTPackContract are being initialized without doing that, and therefore, are prone to incorrect initializations as they may be set to zero addresses or incorrect addresses.

Recommendation

Consider adding appropriate and required checks.

Status

Fixed

Dev Comments: The contract will be deployed via BiddersRewardsFactory, which contains required checks.

K7. Not enough tracking space to accommodate all the NFT IDs

```
27      struct NftData {
28          uint64 bidCount;
29          uint192 cumulativeBidAmount;
30      }
31
32      struct LevelTracking {
33          uint64 totalBidCount;
34          uint192 totalCumulativeBidAmount;
35      }
36
```

The uint space for bidCount and cumulativeBidAmount for a single NFT Id is considered as uint64 and uint192 respectively, whereas, for level tracking, the same uint space has been considered. However, logically, a level is supposed to have a bigger uint space as it will be accommodating all the NFT IDs of this level.

Recommendation

Consider choosing a larger/appropriate uint space in order to accommodate all the NFT Ids.

Status

Acknowledged

Dev Comments: “Having 10^19 bids in one month is a very good problem to have!”

Informational Issues

K8. Logical Code Optimization#1

```
329         for (uint256 i = 0; i < nftLevelIdsListLength; i++) {
330             uint256 nftLevelId;
331             uint256 rewardsMultiplier;
332
333             nftLevelId = nftLevelIdsList[i];
334             rewardsMultiplier = nftTrackerContract.rewardsMultipliers(
335                 nftLevelId
336             );
337
338             require(rewardsMultiplier > 0, "set all rewards multipliers");
339
```

Functions getCombinedLevelsWeighting and releaseFunds calculates weightage of every level as:

rewardsMultiplier * (totalBidCount * COUNT_WEIGHT + totalCumulativeBidAmount * CUM_AMOUNT_WEIGHT)

However, there is a strict condition, that rewardsMultiplier should be set for all the levels, and if any level's rewardsMultiplier is not set, the entire function will revert.

This can be optimized by making another memory array at the start, with a length same as nftLevelIdsListLength, and fetching and storing all the levels' rewardsMultiplier. If any rewardsMultiplier is missing, the function can revert at this stage, without calculating the weightage of any level, thus reducing the number of operations and saving gas cost.

Example Scenario

Let's assume there exists 4 levels, and the authorized roles for any reason missed to set rewardsMultiplier for level 4. The concerned functions will do all the operations and calculate level weightings for levels 1-3, but since level 4 doesn't have any rewardsMultiplier, all the operations will be reverted.

Recommendation

Consider optimizing the code in order to reduce the number of operations and saving gas cost in some scenarios.

Status

Acknowledged

K9. Logical Code Optimization#2

```
340             LevelTracking memory levelTracking = levelTrackingMap[nftLevelId];
341
342             uint64 totalBidCount = levelTracking.totalBidCount;
343             uint192 totalCumulativeBidAmount = levelTracking
344                 .totalCumulativeBidAmount;
345
346             uint256 levelWeighting = rewardsMultiplier *
347                 (totalBidCount *
348                     COUNT_WEIGHT +
349                     totalCumulativeBidAmount *
350                     CUM_AMOUNT_WEIGHT);
351
352             levelWeightings[i] = levelWeighting;
353
354             weightingSum += levelWeighting;
```

Functions getCombinedLevelsWeighting and releaseFunds calculates weightage of every level as:

rewardsMultiplier * (totalBidCount * COUNT_WEIGHT + totalCumulativeBidAmount * CUM_AMOUNT_WEIGHT)

However, the code can be optimized to calculate level weighting only if there exists any bid i.e., totalBidCount > 0, as it doesn't make any sense to calculate level weighting if there exists no bid as the calculation will yield to 0 anyways.

Recommendation

Consider optimizing the code in order to reduce the number of operations and saving gas cost in some scenarios.

Status

Acknowledged

K10. Possibly incorrect Error Statement

```
448     function harvest(address to) public {
449         require(isReleased == true, "funds must be processed");
450         require(to != address(0), "receiver cannot be zero address");
451
452         uint256 tokenId;
453         uint256 userParticipationUnits;
454
```

Harvesting of rewards are supposed to be done once the funds are released. Hence, the statement may be changed to state that the “funds must be released” in order to avoid any confusions.

Example Scenario

Let’s assume, the funds are processed(readyFunds have been called and BiddersRewards contract has been rotated), i.e now there is a new BiddersRewards contract and user may harvest their rewards from the previous rewards contract. But, the funds are not yet released in the previous rewards contract. If a user at this stage comes in to harvest its rewards, it will be notified that the funds are not yet processed and it may assume, that the rotation of rewards contract still yet to happen, but in reality, it has already been done, and only the funds are supposed to be released.

Recommendation

Consider reviewing the error statement and opting for the correct one in order to avoid confusions.

Status

Fixed

K11. Unsafe Downcasting from uint

Unsafe downcasting is done from uint256 to uint128 on line: 371 in releaseFunds() function which can result in undesired results or bugs.

```
368         for (uint256 i = 0; i < nftLevelIdsListLength; i++) {
369             // do rewards proportions per level
370             // note: reward should have 1e18 token decimal factor for division for this expression to be acceptable
371             poolInfos[nftLevelIdsList[i]].reward = uint128(
372                 (totalRewardsBalance * levelWeightings[i]) / weightingSum
373             );
374         }
375
```

Also unsafe downcasting has been done in _harvest() function from uint256 to uint128 on line: 490 which can result in undesired results or bugs.

```
488
489         yoloTokenContract.transfer(to, rewardsYolo);
490         levelPoolInfo.totalPaidOut += uint128(rewardsYolo);
491         emit Harvest(msg.sender, rewardsYolo);
```

Recommendation

Use libraries such as Openzeppelin's `SafeCast` which safely downcasts and reverts the transaction when such an operation overflows. *Reference*

Status

Acknowledged

Dev Comments: "This would imply 10^30 USD in rewards are available in the contract!"

L. BiddersRewardsFactory

High Severity Issues

L1. If the funds are already released, any token upgrade thereafter may burn all the rewards.

```
175         if (
176             nextLevelId.isSemiFungibleBaseType() ||
177             nextLevelId.isNonFungibleBaseType()
178         ) {
179             _mintNFT(sender, nextLevelId, EMPTY_STR);
180         } else {
181             revert("improper nextLevelId encoding");
182         }
183
184         uint256 index = maxIndexes[nextLevelId];
185         uint256 newTokenId = nextLevelId | index;
186
187         // if rewards contract is updated, user should call harvest on old rewards to get remaining amounts BEFORE calling this
188         if (address(biddersRewardsContract) != address(0)) {
189             biddersRewardsContract.bumpDuringUpgrade(id, newTokenId);
190         }
191
192         emit TokenUpgrade(baseType, sender, nextLevelId);
193     }
```

While upgrading a token, YoloNFTPack transfers participation units and level weightings from the old token Id to the new token Id. However, if the funds are released or the rewarder contract has been rotated, it means, the rewarder contract in YoloNFTPack has now been replaced with the new rewarder address, as a result, upgrading tokens at this point will call bumpDuringUpgrade of the new rewarder contract, which doesn't contain any participation units for the old id of this user. If no participation units are being transferred, that means, a user can't harvest rewards for the new token Id that it has upgraded its old token to. Also, as the user has upgraded its token, the old id has now been burned, i.e the rewards generated from the old id can not be extracted as well.

Also, this opportunity could have been utilized to distribute rewards to the remaining participants. However, as the bumpDuringUpgrade was never called for this id on old rewarder contract, the total level weighting will still point to the same weighting, and a portion of the reward will remain unutilized.

Example Scenario

Let's assume, there exist two NFT IDs as N1 and N2 in level 1, and the funds have been released(rewarder contract has been rotated). Let's assume a reward of 500 yolo tokens is supposed to be distributed to the two token IDs as 250 each.

However, the user with token id N2 upgrades its token to UP_N2, may be, in hope to extract larger rewards, due to this upgrade. But since the rewarder contract has been rotated, bumpDuringUpgrade will have no effect or in other words, will not transfer participation units from N2 to UP_N2, as a result the user will not get any rewards by upgrading its token, and will lose its intended rewards from N2 as well, i.e, 250, as the id N2 has now been burned.

Also, this opportunity could have been utilized to distribute all the 500 rewards to the remaining participant that is N1. But since bumpDuringUpgrade was never called on the old rewarder contract, the N1 will still be receiving 250 as reward and remaining 250 will remain unutilized.

Recommendation

Consider reviewing and verifying the business and operational logic, and implementing an appropriate method to notify users to upgrade their tokens(if they want to) before rotating rewards.

Status

Fixed

M. WhitelistSFTClaims

High Severity Issues

M1. After expiration time,user does not get any NFT but still has to pay 1 USDC

```
94      function claimNft() external {
95          address sender = msg.sender;
96          uint256 expireTime = claimeesRegister[sender];
97
98          require(expireTime > 0, "invalid claim");
99
100         // confirm decimals is 6 on polygon USDC contract
101         usdcContract.transferFrom(sender, address(this), 10e6);
102
103         if (block.timestamp < expireTime) {
104             // will revert if level at capacity
105             yoloNFTPackContract.mintBaseSFT(sender);
106
107             uint256 tokenIndex = yoloNFTPackContract.maxIndexes(BASE_SFT_ID);
108             uint256 id = BASE_SFT_ID | tokenIndex;
```

In the case where a user's expiretime has crossed, this function still takes 1 USDC from the user. Ideally this should not happen as this results in the user paying for nothing as he does not get any NFT and he still has to pay 1 USDC for the same.

Recommendation

The user should not be allowed to transfer any USDC token to the contract after expireTime

Status

Fixed

Medium Severity Issues

M2. Centralization of withdraw

```
120     function withdrawUSDC(address receiver↑) external onlyAdmin {
121         uint256 contractBalance = usdcContract.balanceOf(address(this));
122
123         usdcContract.transfer(receiver↑, contractBalance);
124
125         emit Withdrawal(receiver↑, contractBalance);
126     }
```

The withdraw function allows the admin to withdraw all the funds from the contract at any point of time. A malicious admin can exploit this and drain the contract of all the funds.

Recommendation

Consider using a multisig wallet for the same.

Status

Acknowledged

N. General Issues

No need to add zero address check for yolo registry contract address at the time of contract initializations, as it is already being checked in RegistrySatellite.

There are no checks to verify that same registry contract address is being used, throughout the project scope. As it is always possible to have multiple yolo registries.

Example Scenario:
Deploy two yolo registries as regA and regB. For contract X's initialization use regA as a registry while for contract Y's initialization use regB, now individual contracts are working with their own registries, due to which many unexpected scenarios may arise.

Scattered Authorized Roles: The contract initializations provide/grant certain authorized/high privileged roles to the deployer. However, the contracts may be deployed by multiple accounts or deployers, due to which different accounts may be having different roles, which may not be desired/expected.

Instances from Hardat Debugging library console.sol have been reported, which are subject to be removed.

The logical requirement of the team/yolo ecosystem to not allow users to have more than 1 token. But the authorized role can mint both NFT and SemiFT or tokens from two different basetypes for a user, thus bypassing the requirement.

The constructor initializations don't consider enough checks for registered contracts while fetching them from yolo registry. A registered contract can be any address, for instance, a contract registered with the identifier YOLO_TOKEN can be any random contract, or two different identifiers can have the same address. The contract registration is subject to be handled with utmost care.

Possible Scenario:
- Deploy Users contract with Fake Token A
- Set the actual token address for YOLO_TOKEN in the registry
- Deploy Liquidity Pool contract with actual Yolo Token

With this, while minting shares the Liquidity Pool will be depositing actual yolo token from users, but while burning shares, it provide the Fake token registered in Users and not the one deposited by the user i.e, actual yolo token.

BiddersRewards contracts are supposed to be deployed via BiddersRewardsFactory, which takes care of the epoch, to pass an incremented counter for every new contract. However, it should be noted that BiddersRewards may be deployed without the factory, and any epoch can be passed at the time of contract initialization, which may lead to epoch collisions with an existing epoch, and if there exist any offline monitoring tools to track the epoch by emitted events, they may receive the same epoch from two different BiddersRewards contracts.

Make sure to maintain a minimum value for bidding, otherwise, the users may spam the bids in order to harvest more than intended rewards by increasing the bidCount and cumulativeBidAmount of a token id, thus increasing the participation units and also increasing the totalBidCount and totalCumulativeBidAmount, in order to increase the level weighting.

Concerned Scenario:

What if a user is allowed to bid a 0 amount? => It will not take any amount from the user, but still will update tracking details, thus increasing the bidCount of token ID and totalBidCount of the level.

Releasing funds is a critical and one-time operation in BiddersRewards, which calculates level rewards based on the amount of Yolo Tokens available. Hence, it is required to call releaseFunds only after making sure that the contract contains enough/desired yolo liquidity in order to generate intended rewards for every level. A check may be added in releaseFunds to check whether the contract contains the desired liquidity to release funds or not, for instance, yoloTokenBalance >= ExpectedBal. This ExpectedBal may be initialized dynamically for every new contract initialization of BiddersRewards via BiddersRewardsFactory.

The intended operational logic for BiddersRewards is as follows:
Call rotateRewardsContracts in order to process funds for the old rewarder contract(set isReadyToProcess to true) and set a new rewarder contract in NFTTracker and YoloNFTPack
Call releaseFunds in the old rewarder contract and allow participants to harvest rewards.

Funds can be released in two conditions, either the isReadyToProcess has been set to true(by rotating rewards contract), or time duration of 30 days has been passed, which means due to the second condition funds can still be released without the rewards contract being rotated. It may lead to the same vectors mentioned in K3

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the YoloRekt. We performed our audit according to the procedure described above.

Some issues of Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture. In the end, most of the Issues has been Fixed and Acknowledged by the Yolorekt Team.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the YoloRekt Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the YoloRekt Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

500+
Audits Completed

$15B
Secured

500K
Lines of Code Audited

Follow Our Journey

Audit Report
August, 2022

For

yolorekt

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com