SECURITY AUDIT OF

POLYROLL YIELD FARM SMART CONTRACTS

PUBLIC REPORT

AUGUST 05, 2021

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ACRONYMS AND ABBREVIATIONS

NAME	DESCRIPTION
`Ethereum`	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
`Ether (ETH)`	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network
`Smart contract`	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
`Solidity`	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
`Solc`	A compiler for Solidity.
`EVM`	Ethereum Virtual Machine
`LP`	Liquidity Provider

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on July 30, 2021, updated on August 05, 2021. We would like to thank the Polyroll team for trusting Verichains Lab in audit smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Polyroll core smart contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified several vulnerable issues in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Polyroll

Polyroll is a decentralized casino built on Polygon Network and Chainlink. Unlike traditional casinos that operate in black boxes, Polyroll runs on smart contracts that are fair, transparent and immutable.

Polyroll is conducting a fair launch of its native token, ROLL, through yield farming. There will be no pre-sale, no private round VCs, and no pre-mine.

More information at https://polyroll.org/.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the core smart contracts of Polyroll Yield Farm. It was conducted on commit `02403ae2cb0dbd083bc6bf163d8ae00d11017977` of branch `master` from GitHub repository of Polyroll Yield Farm (https://github.com/polyroll/polyroll-farm).

Repository URL of the commit to be audited: https://github.com/polyroll/polyroll-farm/commit/02403ae2cb0dbd083bc6bf163d8ae00d11017977.

The updated version of report is based on the feedback of Polyroll team on July 31, 2021.

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy

- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in Table 1, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1: Vulnerability severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Summary

There are three interfaces and six contracts in the Polyroll Yield Farm source codes:

- `IMasterChef (libs/IMasterChef.sol)`: this is the interface for master chef contract which is used in the `RandGen` contract.
- `IRandGen (libs/IRandGen.sol)`: this is the interface for random generator contract which is used in the `MasterChef` contract.
- `IReferral (libs/IReferral.sol)`: this is the interface for referral contract which is used in the `MasterChef` contract.
- `RollToken (RollToken.sol)`: this is contract for Polyroll Token, which is a customized ERC20 token with hard-coded maximum supply of
- `Prize (Prize.sol)`: this is the contract which will hold and distribute the prizes for lottery games in MasterChef contract.
- `RandGen (RandGen.sol)`: this is the contract that governs the requesting and receiving of random number from Chainlink VRF oracle.
- `Referral (Referral.sol)`: this is the contract which stored the referral records that was used in MasterChef contract to pay referral commissions.
- `Timelock (Timelock.sol)`: this is a timelock contract that is used to support MasterChef contract. This can lock the functionality of MasterChef for a certain amount of time.
- `MasterChef (MasterChef.sol)`: this is the smart contract that handles functionals in Polyroll Yield Farm.

Table 2 shows the summary of findings and their statuses.

#	SUMMARY	SEVERITY	STATUS	UPDATED AT
1	MasterChef – Using deflationary token as LP token will lead to many vulnerabilities	MEDIUM	i Acknowledged	August 05, 2021
2	MasterChef – Missing validation for pool id in *`set`* function	MEDIUM	i Acknowledged	August 05, 2021
3	MasterChef – Misleading comments	LOW	i Acknowledged	August 05, 2021
4	MasterChef – Redundant require statement in the *`updateEmissionRate`* function	LOW	i Acknowledged	August 05, 2021
5	RollToken – Missing delegates updating when transfer tokens	CRITICAL	i Acknowledged	August 05, 2021
6	RollToken – Integer overflow in *`mint`* function	LOW	i Acknowledged	August 05, 2021

7	RollToken – Redundant if statement in `mint` function	**LOW**	ℹ Acknowledged	August 05, 2021
8	RollToken – Misleading comment of `delegates` function	**LOW**	ℹ Acknowledged	August 05, 2021
9	Timelock – Missing checking for valid state when cancel transaction	**LOW**	ℹ Acknowledged	August 05, 2021
10	Timelock – Deprecated `call.value(…)`	**LOW**	ℹ Acknowledged	August 05, 2021
11	RandGen – Multiple randomness requests can lead to result loss	**HIGH**	ℹ Acknowledged	August 05, 2021

Table 2: Findings summary

2.2. Findings

This section contains a detailed analysis of all the vulnerabilities that were discovered by the audit team during the audit process.

2.2.1. MasterChef – Using deflationary token as LP token will lead to many vulnerabilities [MEDIUM]

Consider the following statements in the *deposit* function of MasterChef contract:

```
225  pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
...
234  // Update user's deposit after subtracting fee
235  user.amount = user.amount.add(_amount).sub(depositFee);
```

The statement at line 225 calls the *safeTransferFrom* in *lpToken* contract to transfer *_amount* LP tokens from *msg.sender* to the MasterChef address. Usually, the token transfer has no token fee, so the MasterChef address will receive all *_amount* tokens from *msg.sender*. However, if the LP token is a deflationary token (token with transfer fee), then the MasterChef address will not receive all *_amount* tokens, which would cause many problems for the MasterChef contract. For example, it will make the *user.amount* (as in line 235) to be greater than the actual deposited amount (the amount MasterChef received), so deposited users will get more reward tokens than they should. The *user.amount* variable also affects the referral commission, so the users will get more referral commission.

```
218  uint256 pending =
     user.amount.mul(pool.accRollPerShare).div(1e18).sub(user.rewardDebt);
219  if (pending > 0) {
220    safeRollTransfer(msg.sender, pending);
221    payReferralCommission(msg.sender, pending);
222  }


...

353  uint256 commissionAmount = _pending.mul(referralCommissionRate).div(10000);
354
355  if (referrer != address(0) && commissionAmount > 0) {
356    roll.mint(_user, commissionAmount);
357    roll.mint(referrer, commissionAmount);
358    emit ReferralCommissionPaid(_user, referrer, commissionAmount);
359  }
```

Therefore, if a user keeps depositing and then withdraw again and again, his Polyroll token balance will keep increasing, and the balance of MasterChef in LP token will be drained to approximately zero, which makes other users could not withdraw.

RECOMMENDATION

The current implementation of `MasterChef` contract does not support deflationary token, so before adding a new LP token into `MasterChef` for farming, this LP token must be checked to ensure that it is not a deflationary token.

In addition, to support adding deflationary token, the MasterChef contract must add safety checks to ensure that the total recorded amount of all users in a pool (sum of all *`user.amount`*) will be equal to the balance of MasterChef in that pool's LP token. For example, the contract can use the following code to get the actual deposited amount:

```
uint256 beforeAmount = pool.lpToken.balanceOf(address(this));
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
uint256 actualAmount = pool.lpToken.balanceOf(address(this)).sub(beforeAmount);
```

Then use the *`actualAmount`* value instead of *`_amount`* for calculating.

UPDATES

Polyroll team confirmed that Polyroll does not have deflationary tokens and they have no plans to support it in the future. However, as an auditor, the audit team still leave this finding here as a warning and set this finding's severity to **MEDIUM**.

2.2.2. MasterChef – Missing validation for pool id in `set` function [MEDIUM]

Consider the following code in *`set`* function:

```
function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP) external
onlyOwner {
  require(_depositFeeBP <= 10000, "set: invalid deposit fee basis points");
  totalAllocPoint =
totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
  poolInfo[_pid].allocPoint = _allocPoint;
  poolInfo[_pid].depositFeeBP = _depositFeeBP;
}
```

In the above code, the input pool id *`_pid`* is not validated, so when the pool id is not exists (i.e. *`_pid >= poolInfoLength`*), the contract still updates the value of *`totalAllocPoint`* and *`poolInfo`*, that leads to wrong results in future reward calculations.

RECOMMENDATION

The pool id should be validated as in the following code:

```
function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP) external onlyOwner {
  require(_pid < poolInfoLength, "set: invalid pool id");
  require(_depositFeeBP <= 10000, "set: invalid deposit fee basis points");
  totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
  poolInfo[_pid].allocPoint = _allocPoint;
  poolInfo[_pid].depositFeeBP = _depositFeeBP;
}
```

UPDATES

This finding has been acknowledged by Polyroll team.

2.2.3. MasterChef – Misleading comments [LOW]

The *`randGen`* variable and *`getPrizeRound`* function are having misleading comments.

```
// Roll referral contract address.
IRandGen public randGen;


// View function to see number of players in pool
function getPrizeRound(uint256 _pid) external view returns (uint32) {
  PoolInfo storage pool = poolInfo[_pid];
  return pool.prizeRound;
}
```

RECOMMENDATION

Update the comments to match the variable and function usages.

UPDATES

This finding has been acknowledged by Polyroll team.

2.2.4. MasterChef – Redundant `require` statement in the `updateEmissionRate` function [LOW]

Consider the following code in the *`updateEmissionRate`* function:

```
319  require(rollPerBlock > FINAL_EMISSION_RATE, "updateEmissionRate: Emission rate has
     reached FINAL_EMISSION_RATE already");
...
323  require(currentIndex > lastReductionPeriodIndex, "updateEmissionRate: Wait at least
     1 day after previous update.");
324
325  // Compute new emission rate
326  uint256 newEmissionRate = rollPerBlock;
327  for (uint256 index = lastReductionPeriodIndex; index < currentIndex; ++index) {
328    newEmissionRate = newEmissionRate.mul(1e4 -
     EMISSION_REDUCTION_RATE_PER_PERIOD).div(1e4);
329  }
330  newEmissionRate = newEmissionRate < FINAL_EMISSION_RATE ? FINAL_EMISSION_RATE :
     newEmissionRate;
331  require(newEmissionRate < rollPerBlock, "updateEmissionRate: New emission rate must
     be less than current emission rate.");
```

The *`require`* statement at line 331 is not necessary because *`newEmissionRate`* has an initial value equal to *`rollPerBlock`*, and all the calculations at line 328 (atleast one will be executed) just reduce the *`newEmissionRate`* value.

RECOMMENDATION

Remove that *`require`* statement to make the code cleaner and save gas.

UPDATES

This finding has been acknowledged by Polyroll team.

2.2.5. RollToken – Missing delegates updating when transfer tokens [CRITICAL]

RollToken allows token holders to give voting power to a delegatee. However, with the current implementation of RollToken contract, when the token holder transferred his token from his wallet, voting power of the delegate still remains the same instead of being decreased. That could lead to inflation of voting powers.

Consider the following scenario:

- Wallet A has 10 tokens in balance and don't have any voting power yet.
- A transfers 10 tokens to an empty wallet B. Now B has 10 tokens.
- B delegates its voting power to A by calling the *`delegate`* function. Now A has voting power equivalent to 10 tokens.
- B transfers 10 tokens to C. Now C has 10 tokens; B don't have any tokens but A still has voting power of 10 tokens (because the voting powers were not affected while transferring tokens in the current contract implementation).
- C delegates his voting power to A. Now A has voting power equivalent to 20 tokens (10 from B and 10 from C) while the total balances of A, B and C is only 10 tokens.

By repeating the above process, a user can keep increasing his voting power with an only small initial token balance.

RECOMMENDATION

The contract must update the voting power records when transfering tokens, i.e., when *transfer* and *transferFrom* function are called. One suggestion is to call the *_moveDelegates* function in the *_beforeTokenTransfer* hook of current ERC20 contract implementation as below:

```
function _beforeTokenTransfer(address from, address to, uint256 amount) internal override {
  _moveDelegates(_delegates[from], _delegates[to], amount);
}
```

Note that if the *_moveDelegates* function is called in the *_beforeTokenTransfer* hook, then it must not be called in the *mint* function in `RollToken` contract, since the *_beforeTokenTransfer* hook is already called in the internal *_mint* function.

UPDATES

This finding has been acknowledged by Polyroll team, as in their feedback:

"Acknowledged. We are not using voting function on the ERC20 token and do not plan on doing so in future."

2.2.6. RollToken – Integer overflow in `mint` function [LOW]

Before Solidity version 0.8.0, by default, all arithmetic operations are not checked for integer overflow and underflow, so the contract must handle that itself.

Consider the following code in *mint* function:

```
if (_totalSupply + _amount <= MAX_SUPPLY) {
  _mint(_to, _amount);
  _moveDelegates(address(0), _delegates[_to], _amount);
} else if (_totalSupply + _amount > MAX_SUPPLY && _totalSupply <= MAX_SUPPLY) {
  _mint(_to, MAX_SUPPLY - _totalSupply);
  _moveDelegates(address(0), _delegates[_to], MAX_SUPPLY - _totalSupply);
}
```

An overflow can occur in the *_totalSupply + _amount* addition and leads to wrong result for the condition in the checking statement. For example, if *_totalSupply=1000*, *_amount*=$2^{256} - 1$ (the maximum value for *uint256* in Solidity), then the *_totalSupply + _amount* addition will return *999*, which is obviously not the result we want.

RECOMMENDATION

A simple solution is to check for integer overflow when calculate the new total supply, as in the following code:

```
uint _totalSupply = totalSupply();
require(_totalSupply < MAX_SUPPLY, "Total supply must not exceed max supply.");

uint256 _newSupply = _totalSupply + _amount;
require(_newSupply > _totalSupply, "Integer addition overflow.");

// Impose maximum total Supply on minting of token.
if (_newSupply <= MAX_SUPPLY) {
  _mint(_to, _amount);
  _moveDelegates(address(0), _delegates[_to], _amount);
} else if (_newSupply > MAX_SUPPLY && _totalSupply <= MAX_SUPPLY) {
  _mint(_to, MAX_SUPPLY - _totalSupply);
  _moveDelegates(address(0), _delegates[_to], MAX_SUPPLY - _totalSupply);
}
```

Another solution is to use the OpenZeppelin's `SafeMath` library [1] for arithmetic operations. If any integer overflow/underflow occurs, the transaction will be reverted. Note that from Solidity version 0.8.0, arithmetic operations will be reverted on underflow and overflow by default[2] without using *`SafeMath`* library.

UPDATES

This finding has been acknowledged by Polyroll team.

2.2.7. RollToken – Redundant `if` statement in `mint` function [LOW]

Consider the following code of the *`mint`* function:

```
function mint(address _to, uint256 _amount) public onlyOwner {
  uint _totalSupply = totalSupply();
  require(_totalSupply < MAX_SUPPLY, "Total supply must not exceed max supply.");

  // Impose maximum total Supply on minting of token.
  if (_totalSupply + _amount <= MAX_SUPPLY) {
    _mint(_to, _amount);
    _moveDelegates(address(0), _delegates[_to], _amount);
  } else if (_totalSupply + _amount > MAX_SUPPLY && _totalSupply <= MAX_SUPPLY) {
    _mint(_to, MAX_SUPPLY - _totalSupply);
    _moveDelegates(address(0), _delegates[_to], MAX_SUPPLY - _totalSupply);
  }

}
```

[1] https://docs.openzeppelin.com/contracts/3.x/api/math
[2] https://docs.soliditylang.org/en/v0.8.0/080-breaking-changes.html

At line 20, the *`if`* statement has two conditions:

- *`_totalSupply + _amount > MAX_SUPPLY`*: this is always *`true`*, because in this case the condition at line 17 is already *`false`*.
- *`_totalSupply <= MAX_SUPPLY`*: this is always *`true`*, because in this case the condition at line 14 is already satisfied.

So, these two conditions are already satisfied, hence the checking is not necessary and can be removed to save gas and make the code cleaner.

RECOMMENDATION

Remove the redundant `if` statement as below:

```
17  if (_totalSupply + _amount <= MAX_SUPPLY) {
18    _mint(_to, _amount);
19    _moveDelegates(address(0), _delegates[_to], _amount);
20  } else {
21    _mint(_to, MAX_SUPPLY - _totalSupply);
22    _moveDelegates(address(0), _delegates[_to], MAX_SUPPLY - _totalSupply);
23  }
```

UPDATES

This finding has been acknowledged by Polyroll team.

2.2.8. RollToken – Misleading comment of `delegates` function [LOW]

The function *`delegates`* in the `RollToken` contract is used for getting the delegatee address of *`delegator`*, not for delegating votes.

```
/**
  * @dev Delegate votes from `msg.sender` to `delegatee`
  * @param delegator The address to get delegatee for
  */
function delegates(address delegator)
  ...
```

RECOMMENDATION

Update the comment to match the function’s usage.

UPDATES

This finding has been acknowledged by Polyroll team.

2.2.9. Timelock – Missing checking for valid state when cancel transaction [LOW]

A transaction should not be canceled if it is not in queue. Look at the current implementation of *`cancelTransaction`* function:

```
function cancelTransaction(address target, uint value, string memory signature, bytes memory data, uint eta) public {
  require(msg.sender == admin, "Timelock::cancelTransaction: Call must come from admin.");

  bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
  queuedTransactions[txHash] = false;

  emit CancelTransaction(txHash, target, value, signature, data, eta);
}
```

If the *cancelTransaction* was called by `admin`, the *CancelTransaction* event will be emitted for every input transaction, even if that transaction has been executed or has not been queued yet. This behavior may lead to confusion for contract's event listeners, and wasting gas for unnecessary event emitting.

RECOMMENDATION

Only cancel transactions that are currently in queue.

```
function cancelTransaction(address target, uint value, string memory signature, bytes memory data, uint eta) public {
  require(msg.sender == admin, "Timelock::cancelTransaction: Call must come from admin.");

  bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
  require(queuedTransactions[txHash], "Timelock::cancelTransaction: Transaction must be queued.");
  queuedTransactions[txHash] = false;

  emit CancelTransaction(txHash, target, value, signature, data, eta);
}
```

UPDATES

This finding has been acknowledged by Polyroll team.

2.2.10. Timelock – Deprecated `call.value(…)` *[LOW]*

The following code is from the executeTransaction function in Timelock contract:

```
(bool success, bytes memory returnData) = target.call.value(value)(callData);
```

From Solidity version 0.6.4, the syntax *call.value(…)* was deprecated in favor of *call{value: …}*[3]. The contract should use the new syntax to avoid potential issues in future.

[3] https://github.com/ethereum/solidity/releases/tag/v0.6.4

RECOMMENDATION

Replace the deprecated code with the new one as below:

```
(bool success, bytes memory returnData) = target.call{value: value}(callData);
```

UPDATES

This finding has been acknowledged by Polyroll team.

2.2.11. RandGen – Multiple randomness requests can lead to result loss [HIGH]

Current implementation of the *RandGen* contract can only store one randomness request result at a moment. Therefore, if we call the *getRandomNumber* method multiple times, whenever receiving a *fulfillRandomness* callback, the contract will override the previous result with the new result. Moreover, if the contract has multiple randomness requests in flight simultaneously, the order in which the fulfillments arrive cannot be ensured.

RECOMMENDATION

The contract should use mapping to store the random number results for each randomness request. The *settleLoterry* function needs also to be modified to accept request id as an input parameter.

```
...  mapping (bytes32 => bool) private _isFulfilled;
     mapping (bytes32 => uint) private _results;

// Callback function used by VRF Coordinator
function fulfillRandomness(bytes32 _requestId, uint _randomness) internal override
{
  _isFulfilled[requestId] = true;
  _results[requestId] = randomness;
  emit RandomnessFulfilled(_requestId, _randomness);
}

// SettleLottery function is seperated from fulfillRandomness because
// Chainlink VRF has a very low gas limit and we need to call settleLottery and pay
gas fees ourselves.
function settleLottery(bytes32 _requestId) external onlyOwner {
  require(_isFulfilled[requestId], "Request has not been fulfilled yet.");
  masterChef.settleLottery(requestId, _results[requestId]);
}
```

Note that if the values range of the needed random number is not as large as a *uint256* number, then we could optimize the above code to use only one mapping, thus optimize the gas usage.

UPDATES

This finding has been acknowledged by Polyroll team, as in their feedback:

“Acknowledged. We will be closing and shutting down our lottery farms, so it does not matter any more.”

2.3. Additional notes and recommendations

2.3.1. RandGen – Adding `RandomnessFulfilled` *event*

The *fulfillRandomness* function should emit an event to notify the listeners (e.g. the owner) that the current random generating request has been fulfilled and the owner can call *settleLottery* next.

Note that the Chainlink VRF Corrdinator has very low gas limit (200k gas)[4], so the *fulfillRandomness* callback function must uses less gas than this limit, otherwise the transaction will fail.

RECOMMENDATION

Emit new *RandomnessFulfilled* event in the *fulfillRandomness* function.

```
...
event RandomnessFulfilled(bytes32 _requestId, uint _randomness);

// Callback function used by VRF Coordinator
function fulfillRandomness(bytes32 _requestId, uint _randomness) internal override
{
  requestId = _requestId;
  randomness = _randomness;
  emit RandomnessFulfilled(requestId, randomness);
}
```

2.3.2. MasterChef – Effectively manage players in a pool

In the current implementation of `MasterChef` contract, the struct *PoolInfo* uses two arrays (*pool.players*, *pool.playerStakes*) and one mapping (*pool.playerIds*) to store and manage players.

```
struct PoolInfo {
...
  ...
  address[] players;          // List of addresses of lottery players.
  uint256[] playerStakes;     // List of player's stakes a.k.a. player's contribution to lottery pool.
  mapping(address => uint256) playerIds; // Mapping of address to indexes. Used to check if player has participated before this round.
}
```

However, that is not the most effective way to do it. The *playerIds* array is only used for checking the existance of a player. Instead, the contract could combine the *playerStakes* array and *playerIds* mapping into one *(address => uint256) playerStakes* mapping: if the *playerStakes[A]* is zero, then the player A has not been added to the pool yet. Note that the contract must reset values from *playerStakes* for each new round.

[4] https://docs.chain.link/docs/get-a-random-number/#random-number-consumer

Additionally, in the current implementation of *settleLottery* function, after each round, the values of *pool.players* and *pool.playerStakes* will be reset by assign them to new empty array:

```
pool.players = new address[](0);
pool.playerStakes = new uint256[](0);
```

This is not the best way to reset values for each new round, because these empty elements will eventually be filled again, so it is not an efficient gas usage. The best way is simply not to clear them. Instead, the contract should keep a separate *pool.playerCount* and manages the players like the way it manages pools using *poolInfoLength* and *poolInfo* mapping. In this way, the old round players still remain in the pool (before being replaced by new players), but are ignored.

3. VERSION HISTORY

VERSION	DATE	STATUS/CHANGES	CREATED BY
0.1	July 30, 2021	Initial report	Verichains Lab
1.0	August 05, 2021	Private report	Verichains Lab
1.1	August 05, 2021	Public report	Verichains Lab

APPENDIX A: OVERVIEW FUNCTION CALL GRAPH

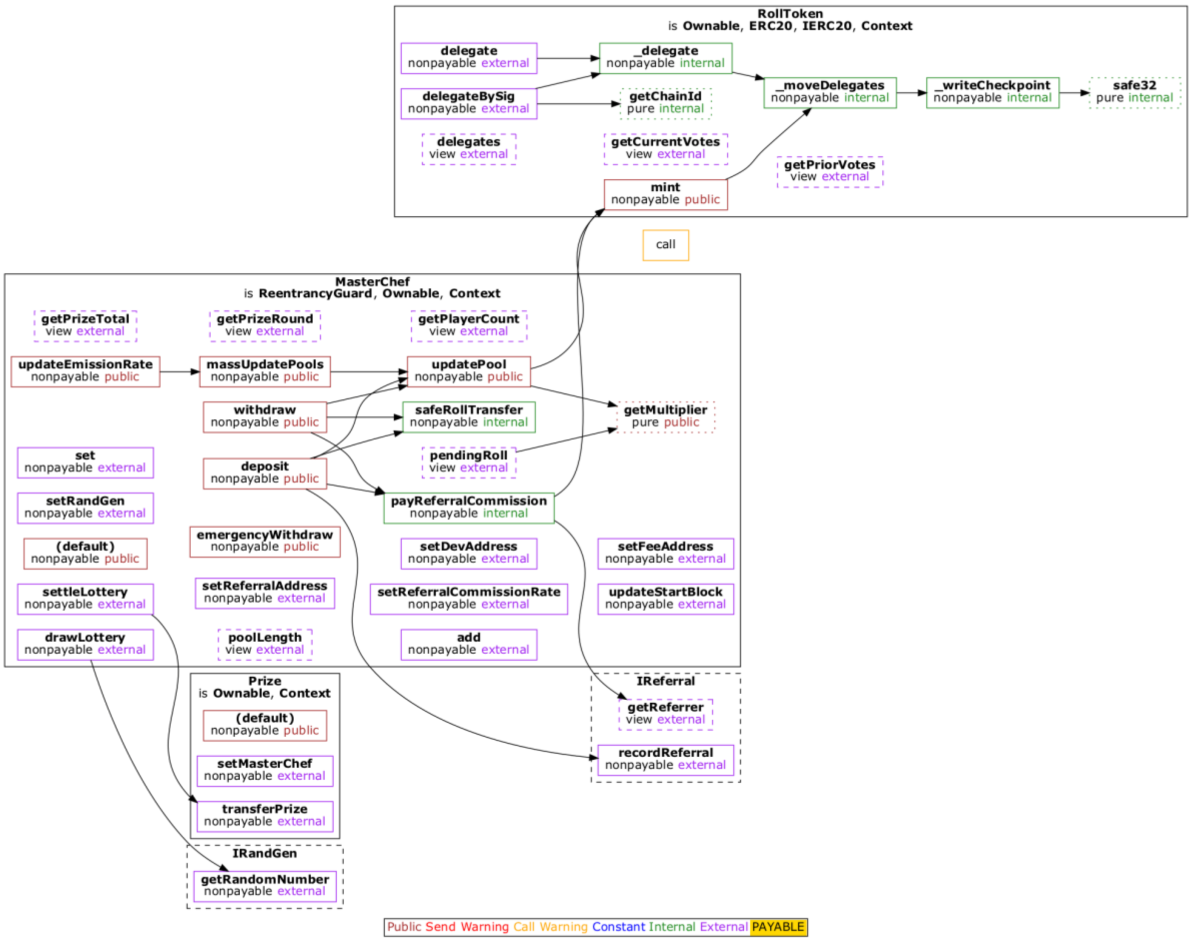
RollToken
is Ownable, ERC20, IERC20, Context
delegate nonpayable external
_delegate nonpayable internal
delegateBySig nonpayable external
getChainId pure internal
_moveDelegates nonpayable internal
_writeCheckpoint nonpayable internal
safe32 pure internal
delegates view external
getCurrentVotes view external
getPriorVotes view external
mint nonpayable public
call
MasterChef
is ReentrancyGuard, Ownable, Context
getPrizeTotal view external
getPrizeRound view external
getPlayerCount view external
updateEmissionRate nonpayable public
massUpdatePools nonpayable public
updatePool nonpayable public
withdraw nonpayable public
safeRollTransfer nonpayable internal
getMultiplier pure public
set nonpayable external
deposit nonpayable public
pendingRoll view external
payReferralCommission nonpayable internal
setRandGen nonpayable external
emergencyWithdraw nonpayable public
(default) nonpayable public
setDevAddress nonpayable external
setFeeAddress nonpayable external
settleLottery nonpayable external
setReferralAddress nonpayable external
setReferralCommissionRate nonpayable external
updateStartBlock nonpayable external
drawLottery nonpayable external
poolLength view external
add nonpayable external
Prize
is Ownable, Context
(default) nonpayable public
setMasterChef nonpayable external
transferPrize nonpayable external
IReferral
getReferrer view external
recordReferral nonpayable external
IRandGen
getRandomNumber nonpayable external
Public Send Warning Call Warning Constant Internal External PAYABLE

Figure 1: function call graph of MasterChef contract

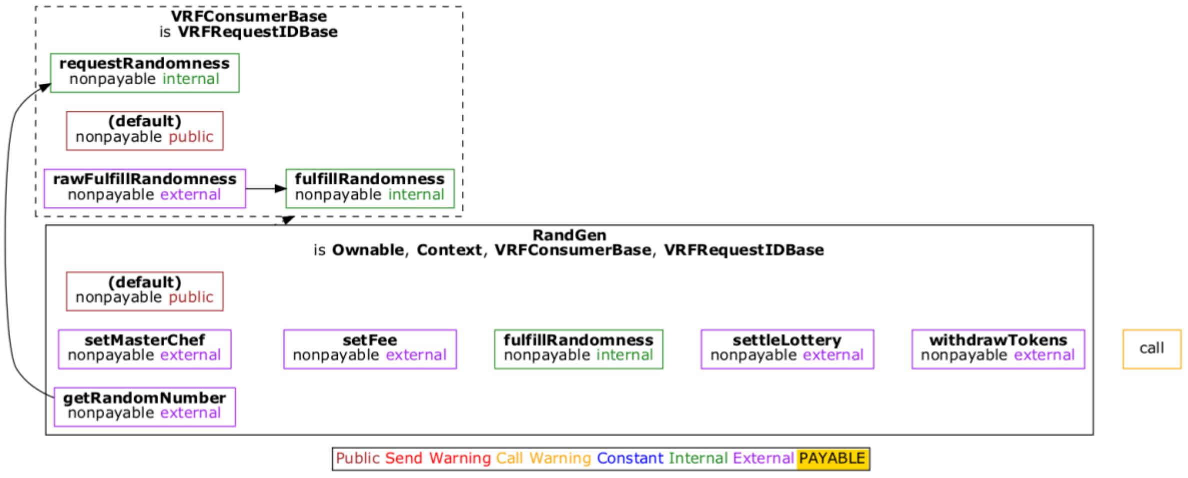
VRFConsumerBase
is VRFRequestIDBase
requestRandomness nonpayable internal
(default) nonpayable public
rawFulfillRandomness nonpayable external
fulfillRandomness nonpayable internal
RandGen
is Ownable, Context, VRFConsumerBase, VRFRequestIDBase
(default) nonpayable public
setMasterChef nonpayable external
setFee nonpayable external
fulfillRandomness nonpayable internal
settleLottery nonpayable external
withdrawTokens nonpayable external
call
getRandomNumber nonpayable external
Public Send Warning Call Warning Constant Internal External PAYABLE

Figure 2: function call graph of RandGen contract

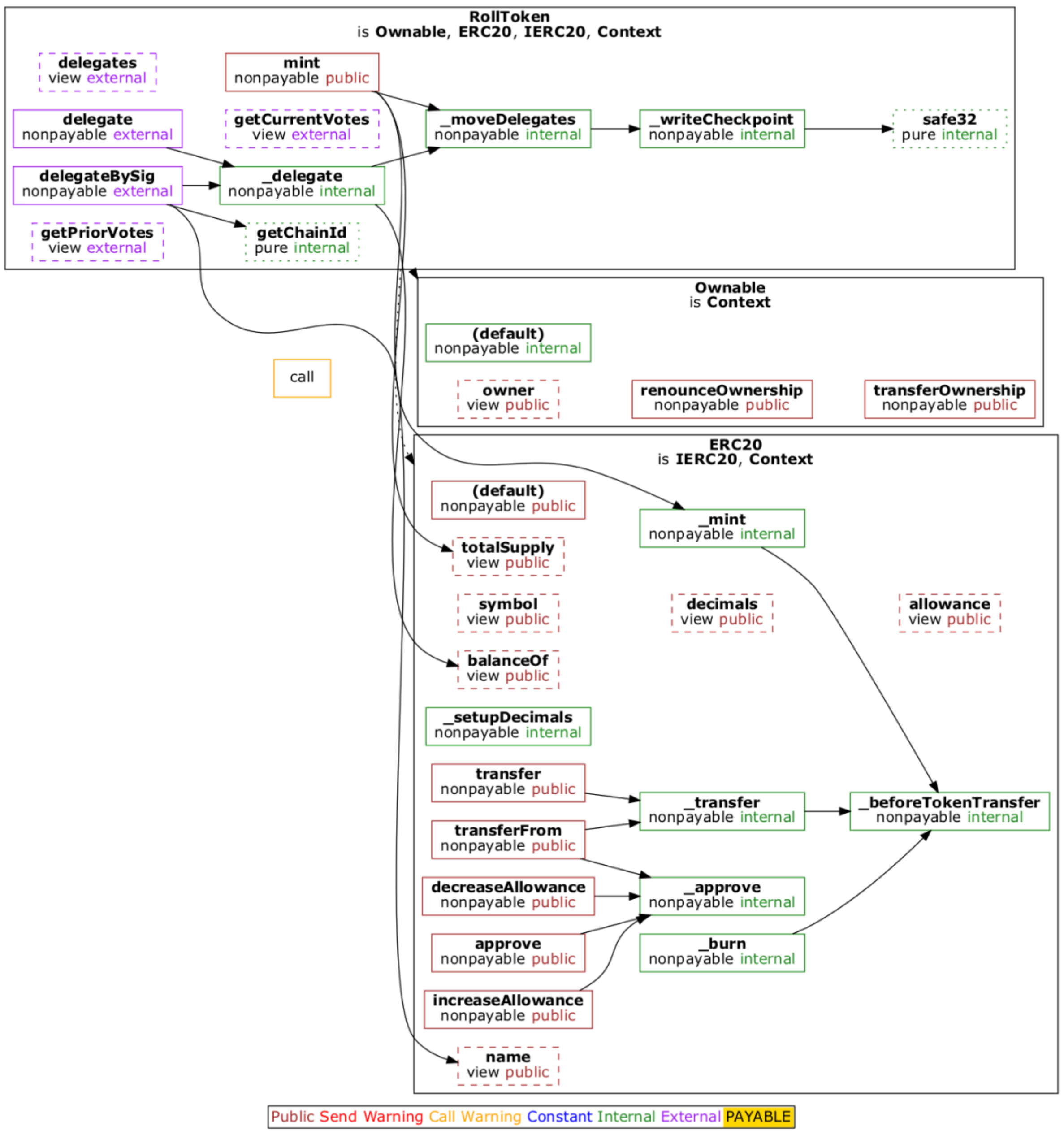
RollToken
is Ownable, ERC20, IERC20, Context
delegates view external
mint nonpayable public
delegate nonpayable external
getCurrentVotes view external
_moveDelegates nonpayable internal
_writeCheckpoint nonpayable internal
safe32 pure internal
delegateBySig nonpayable external
_delegate nonpayable internal
getPriorVotes view external
getChainId pure internal
Ownable
is Context
(default) nonpayable internal
call
owner view public
renounceOwnership nonpayable public
transferOwnership nonpayable public
ERC20
is IERC20, Context
(default) nonpayable public
_mint nonpayable internal
totalSupply view public
symbol view public
decimals view public
allowance view public
balanceOf view public
_setupDecimals nonpayable internal
transfer nonpayable public
_transfer nonpayable internal
_beforeTokenTransfer nonpayable internal
transferFrom nonpayable public
decreaseAllowance nonpayable public
_approve nonpayable internal
approve nonpayable public
_burn nonpayable internal
increaseAllowance nonpayable public
name view public
Public Send Warning Call Warning Constant Internal External PAYABLE

Figure 3: function call graph of RollToken contract