SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	LooksRare
Prepared by:	Sherlock
Lead Security Expert:	cergyk
Dates Audited:	October 30 - November 4, 2023
Prepared on:	November 22, 2023

Introduction

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

Scope

Repository: LooksRare/contracts-infiltration

Branch: master

Commit: 90bd6af2da7b3df1c5fac6595128ab62cf989cca

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

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

Issues found

Medium	High
4	3

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

cergyk
detectiveking
mstpr-brainbot
klaus
SilentDefendersOfDeFi
ast3ros
0xReiAyanami
p-tsanev
ge6a
lil.eth
coffiasd
Krace
pontifex
0xGoodess
Tricko

Issue H-1: _killWoundedAgents

Source: https://github.com/sherlock-audit/2023-10-looksrare-judging/issues/21

Found by

ast3ros, cergyk, klaus, mstpr-brainbot

If an agent healed is wounded again, the agent will die from the previous wound that was healed. The user spends LOOKS tokens to heal and success to heal, but as the result, the agent will die.

Vulnerability Detail

The `_killWoundedAgents` function only checks the status of the agent, not when it was wounded.

```
    function _killWoundedAgents(
        uint256 roundId,
        uint256 currentRoundAgentsAlive
    ) private returns (uint256 deadAgentsCount) {
        ...
        for (uint256 i; i < woundedAgentIdsCount; ) {
            uint256 woundedAgentId = woundedAgentIdsInRound[i.unsafeAdd(1)];

            uint256 index = agentIndex(woundedAgentId);
@>          if (agents[index].status == AgentStatus.Wounded) {
                ...
            }

            ...
        }

        emit Killed(roundId, woundedAgentIds);
    }
```

So when `fulfillRandomWords` kills agents that were wounded and unhealed at round `currentRoundId - ROUNDS_TO_BE_WOUNDED_BEFORE_DEAD`, it will also kill the agent who was healed and wounded again after that round.

Also, since `fulfillRandomWords` first draws the new wounded agents before kills agents, in the worst case scenario, agent could die immediately after being wounded in this round.

```
if (activeAgents > NUMBER_OF_SECONDARY_PRIZE_POOL_WINNERS) {
@>  uint256 woundedAgents = _woundRequestFulfilled(
        currentRoundId,
```

```
        currentRoundAgentsAlive,
        activeAgents,
        currentRandomWord
    );

    uint256 deadAgentsFromKilling;
    if (currentRoundId > ROUNDS_TO_BE_WOUNDED_BEFORE_DEAD) {
@>      deadAgentsFromKilling = _killWoundedAgents({
            roundId:
↪   currentRoundId.unsafeSubtract(ROUNDS_TO_BE_WOUNDED_BEFORE_DEAD),
            currentRoundAgentsAlive: currentRoundAgentsAlive
        });
    }
```

This is the PoC test code. You can add it to the Infiltration.fulfillRandomWords.t.sol file and run it.

```
function test_poc() public {

    _startGameAndDrawOneRound();

    uint256[] memory randomWords = _randomWords();
    uint256[] memory woundedAgentIds;

    for (uint256 roundId = 2; roundId <= ROUNDS_TO_BE_WOUNDED_BEFORE_DEAD + 1;
↪   roundId++) {

        if(roundId == 2) { // heal agent. only woundedAgentIds[0] dead.
            (woundedAgentIds, ) = infiltration.getRoundInfo({roundId: 1});
            assertEq(woundedAgentIds.length, 20);

            _drawXRounds(1);

            _heal({roundId: 3, woundedAgentIds: woundedAgentIds});

            _startNewRound();

            // everyone except woundedAgentIds[0] is healed
            uint256 agentIdThatWasKilled = woundedAgentIds[0];

            IInfiltration.HealResult[] memory healResults = new
↪   IInfiltration.HealResult[](20);
            for (uint256 i; i < 20; i++) {
                healResults[i].agentId = woundedAgentIds[i];

                if (woundedAgentIds[i] == agentIdThatWasKilled) {
                    healResults[i].outcome = IInfiltration.HealOutcome.Killed;
                } else {
```

```
                healResults[i].outcome = IInfiltration.HealOutcome.Healed;
            }
        }

        expectEmitCheckAll();
        emit HealRequestFulfilled(3, healResults);

        expectEmitCheckAll();
        emit RoundStarted(4);

        randomWords[0] = (69 * 10_000_000_000) + 9_900_000_000; // survival rate 99%, first one gets killed

        vm.prank(VRF_COORDINATOR);
        VRFConsumerBaseV2(address(infiltration)).rawFulfillRandomWords(_computeVrfRequestId(3), randomWords);

        for (uint256 i; i < woundedAgentIds.length; i++) {
            if (woundedAgentIds[i] != agentIdThatWasKilled) {
                _assertHealedAgent(woundedAgentIds[i]);
            }
        }

        roundId += 2; // round 2, 3 used for healing
    }

    _startNewRound();

    // Just so that each round has different random words
    randomWords[0] += roundId;

    if (roundId == ROUNDS_TO_BE_WOUNDED_BEFORE_DEAD + 1) { // wounded agents at round 1 are healed, only woundedAgentIds[0] was dead.
        (uint256[] memory woundedAgentIdsFromRound, ) = infiltration.getRoundInfo({
            roundId: uint40(roundId - ROUNDS_TO_BE_WOUNDED_BEFORE_DEAD)
        });

        // find re-wounded agent after healed
        uint256[] memory woundedAfterHeal = new uint256[](woundedAgentIds.length);
        uint256 totalWoundedAfterHeal;
        for (uint256 i; i < woundedAgentIds.length; i ++){
            uint256 index = infiltration.agentIndex(woundedAgentIds[i]);
            IInfiltration.Agent memory agent = infiltration.getAgent(index);
            if (agent.status == IInfiltration.AgentStatus.Wounded) {
                woundedAfterHeal[i] = woundedAgentIds[i]; // re-wounded agent will be killed
```

```
                    totalWoundedAfterHeal++;
                }
                else{
                    woundedAfterHeal[i] = 0; // set not wounded again 0
                }

            }
            expectEmitCheckAll();
            emit Killed(roundId - ROUNDS_TO_BE_WOUNDED_BEFORE_DEAD,
↪   woundedAfterHeal);
        }

        expectEmitCheckAll();
        emit RoundStarted(roundId + 1);

        uint256 requestId = _computeVrfRequestId(uint64(roundId));
        vm.prank(VRF_COORDINATOR);

↪   VRFConsumerBaseV2(address(infiltration)).rawFulfillRandomWords(requestId,
↪   randomWords);
    }
}
```

Impact

The user pays tokens to keep the agent alive, but agent will die even if agent success to healed. The user has lost tokens and is forced out of the game.

Code Snippet

https://github.com/sherlock-audit/2023-10-looksrare/blob/86e8a3a6d7880af0dc2ca03bf3eb31bc0a10a552/contracts-infiltration/contracts/Infiltration.sol#L1489

https://github.com/sherlock-audit/2023-10-looksrare/blob/86e8a3a6d7880af0dc2ca03bf3eb31bc0a10a552/contracts-infiltration/contracts/Infiltration.sol#L1130-L1143

Tool used

Manual Review

Recommendation

Check woundedAt at `_killWoundedAgents`

```
    function _killWoundedAgents(
        uint256 roundId,
        uint256 currentRoundAgentsAlive
    ) private returns (uint256 deadAgentsCount) {
        ...
        for (uint256 i; i < woundedAgentIdsCount; ) {
            uint256 woundedAgentId = woundedAgentIdsInRound[i.unsafeAdd(1)];

            uint256 index = agentIndex(woundedAgentId);
-           if (agents[index].status == AgentStatus.Wounded) {
+           if (agents[index].status == AgentStatus.Wounded &&
 ↪  agents[index].woundedAt == roundId) {
                ...
            }

            ...
        }

        emit Killed(roundId, woundedAgentIds);
    }
```

Discussion

0xhiroshi

https://github.com/LooksRare/contracts-infiltration/pull/148

0xhiroshi

https://github.com/LooksRare/contracts-infiltration/pull/164

SergeKireev

Fix LGTM

Issue H-2: Winning agent id may be uninitialized when game is over, locking grand prize

Source: https://github.com/sherlock-audit/2023-10-looksrare-judging/issues/31

Found by

cergyk, detectiveking In the `Infiltration` contract, the `agents` mapping holds all of the agents structs, and encodes the ranking of the agents (used to determine prizes at the end of the game).

This mapping records are lazily initialized when two agents are swapped (an agent is either killed or escapes):

- The removed agent goes to the end of currently alive agents array with the status `Killed/Escaped` and its `agentId` is set
- The last agent of the currently alive agents array is put in place of the previously removed agent and its `agentId` is set

This is the only moment when the `agentId` of an agent record is set.

This means that if the first agent in the array ends up never being swapped, it keeps its agentId as zero, and the grand prize is unclaimable.

Vulnerability Detail

We can see in the implementation of `claimGrandPrize` that:
https://github.com/sherlock-audit/2023-10-looksrare/blob/main/contracts-infiltration/contracts/Infiltration.sol#L658

The field `Agent.agentId` of the struct is used to determine if the caller can claim. Since the id is zero, and it is and invalid id for an agent, there is no owner for it and the condition:

https://github.com/sherlock-audit/2023-10-looksrare/blob/main/contracts-infiltration/contracts/Infiltration.sol#L1666-L1670

Always reverts.

Impact

The grand prize ends up locked/unclaimable by the winner

Code Snippet

Tool used

Manual Review

Recommendation

In `claimGrandPrize` use 1 as the default if `agents[1].agentId == 0`:

```
function claimGrandPrize() external nonReentrant {
    _assertGameOver();
    uint256 agentId = agents[1].agentId;
+   if (agentId == 0)
+       agentId = 1;
    _assertAgentOwnership(agentId);

    uint256 prizePool = gameInfo.prizePool;

    if (prizePool == 0) {
        revert NothingToClaim();
    }

    gameInfo.prizePool = 0;

    _transferETHAndWrapIfFailWithGasLimit(WETH, msg.sender, prizePool,
↪   gasleft());

    emit PrizeClaimed(agentId, address(0), prizePool);
}
```

Discussion

0xhiroshi

https://github.com/LooksRare/contracts-infiltration/pull/153

0xhiroshi

https://github.com/LooksRare/contracts-infiltration/pull/178

@nevillehuang

SergeKireev

Fix LGTM

Issue H-3: Attacker can steal reward of actual winner by force ending the game

Source: https://github.com/sherlock-audit/2023-10-looksrare-judging/issues/98

Found by

0xReiAyanami, SilentDefendersOfDeFi, cergyk, mstpr-brainbot

A malicious user can force win the game by escaping all but one wounded agent, and steal the grand price.

Vulnerability Detail

Currently following scenario is possible: There is an attacker owning some lower index agents and some higher index agents. There is a normal user owing one agent with an index between the attackers agents. If one of the attackers agents with an lower index gets wounded, he can escape all other agents and will instantly win the game, even if the other User has still one active agent.

This is possible because because the winner is determined by the agent index, and escaping all agents at once wont kill the wounded agent because the game instantly ends.

Following check inside startNewRound prevents killing of wounded agents by starting a new round:

```
uint256 activeAgents = gameInfo.activeAgents;
        if (activeAgents == 1) {
            revert GameOver();
        }
```

Following check inside of claimPrize pays price to first ID agent:

```
uint256 agentId = agents[1].agentId;
_assertAgentOwnership(agentId);
```

See following POC:

POC

Put this into Infiltration.mint.t.sol and run `forge test --match-test forceWin -vvv`

```
function test_forceWin() public {
        address attacker = address(1337);
```

```
        //prefund attacker and user1
        vm.deal(user1, PRICE * MAX_MINT_PER_ADDRESS);
        vm.deal(attacker, PRICE * MAX_MINT_PER_ADDRESS);

        // MINT some agents
        vm.warp(_mintStart());
        // attacker wants to make sure he owns a bunch of agents with low IDs!!
        vm.prank(attacker);
        infiltration.mint{value: PRICE * 30}({quantity: 30});
        // For simplicity we mint only 1 agent to user 1 here, but it could be
↪   more, they could get wounded, etc.
        vm.prank(user1);
        infiltration.mint{value: PRICE *1}({quantity: 1});
        //Attacker also wants a bunch of agents with the highest IDs, as they
↪   are getting swapped with the killed agents (move forward)
        vm.prank(attacker);
        infiltration.mint{value: PRICE * 30}({quantity: 30});

        vm.warp(_mintEnd());

        //start the game
        vm.prank(owner);
        infiltration.startGame();

        vm.prank(VRF_COORDINATOR);
        uint256[] memory randomWords = new uint256[](1);
        randomWords[0] = 69_420;
        VRFConsumerBaseV2(address(infiltration)).rawFulfillRandomWords(_computeV
↪   rfRequestId(1), randomWords);
        // Now we are in round 2 we do have 1 wounded agent (but we can imagine
↪   any of our agent got wounded, doesnt really matter)

        // we know with our HARDCODED RANDOMNESS THAT AGENT 3 gets wounded!!

        // Whenever we get in a situation, that we own all active agents, but 1
↪   and our agent has a lower index we can instant win the game!!
        // This is done by escaping all agents, at once, except the lowest index
        uint256[] memory escapeIds = new uint256[](59);
        escapeIds[0] = 1;
        escapeIds[1] = 2;
        uint256 i = 4; //Scipping our wounded AGENT 3
        for(; i < 31;) {
            escapeIds[i-2] = i;
            unchecked {++i;}
        }
        //skipping 31 as this owned by user1
        unchecked {++i;}
        for(; i < 62;) {
```


```
            escapeIds[i-3] = i;
            unchecked {++i;}
        }
        vm.prank(attacker);
        infiltration.escape(escapeIds);

        (uint16 activeAgents, uint16 woundedAgents, , uint16 deadAgents, , , , ,
↪   , , ) = infiltration.gameInfo();
        console.log("Active", activeAgents);
        assertEq(activeAgents, 1);
        // This will end the game instantly.
        //owner should not be able to start new round
        vm.roll(block.number + BLOCKS_PER_ROUND);
        vm.prank(owner);
        vm.expectRevert();
        infiltration.startNewRound();

        //Okay so the game is over, makes sense!
        // Now user1 has the only active AGENT, so he should claim the grand
↪  prize!
        // BUT user1 cannot
        vm.expectRevert(IInfiltration.NotAgentOwner.selector);
        vm.prank(user1);
        infiltration.claimGrandPrize();

        //instead the attacker can:
        vm.prank(attacker);
        infiltration.claimGrandPrize();
```

Impact

Attacker can steal the grand price of the actual winner by force ending the game trough escapes.

This also introduces problems if there are other players with wounded agents but lower < 50 TokenID, they can claim prices for wounded agents, which will break parts of the game logic.

Code Snippet

https://github.com/sherlock-audit/2023-10-looksrare/blob/main/contracts-infiltration/contracts/Infiltration.sol#L589-L592

https://github.com/sherlock-audit/2023-10-looksrare/blob/main/contracts-infiltration/contracts/Infiltration.sol#L656-L672

Tool used

Manual Review

Recommendation

Start a new Round before the real end of game to clear all wounded agents and reorder IDs.

Discussion

0xhiroshi

https://github.com/LooksRare/contracts-infiltration/pull/154

SergeKireev

Fix LGTM

Issue M-1: Agents with Healing Opportunity Will Be Terminated Directly if The `escape` Reduces activeAgents to the Number of `NUMBER_OF_SECONDARY_PRIZE_POOL_WINNERS` or Fewer

Source: https://github.com/sherlock-audit/2023-10-looksrare-judging/issues/43

Found by

Krace, SilentDefendersOfDeFi, detectiveking, mstpr-brainbot

Wounded Agents face the risk of losing their last opportunity to heal and are immediately terminated if certain Active Agents decide to escape.

Vulnerability Detail

In each round, agents have the opportunity to either `escape` or `heal` before the `_requestForRandomness` function is called. However, the order of execution between these two functions is not specified, and anyone can be executed at any time just before `startNewRound`. Typically, this isn't an issue. However, the problem arises when there are only a few Active Agents left in the game.

On one hand, the `heal` function requires that the number of `gameInfo.activeAgents` is greater than `NUMBER_OF_SECONDARY_PRIZE_POOL_WINNERS`.

```
    function heal(uint256[] calldata agentIds) external nonReentrant {
        _assertFrontrunLockIsOff();
//@audit If there are not enough activeAgents, heal is disabled
        if (gameInfo.activeAgents <= NUMBER_OF_SECONDARY_PRIZE_POOL_WINNERS) {
            revert HealingDisabled();
        }
```

On the other hand, the `escape` function will directly set the status of agents to "ESCAPE" and reduce the count of `gameInfo.activeAgents`.

```
function escape(uint256[] calldata agentIds) external nonReentrant {
     _assertFrontrunLockIsOff();

     uint256 agentIdsCount = agentIds.length;
     _assertNotEmptyAgentIdsArrayProvided(agentIdsCount);

     uint256 activeAgents = gameInfo.activeAgents;
     uint256 activeAgentsAfterEscape = activeAgents - agentIdsCount;
     _assertGameIsNotOverAfterEscape(activeAgentsAfterEscape);
```

```
    uint256 currentRoundAgentsAlive = agentsAlive();

    uint256 prizePool = gameInfo.prizePool;
    uint256 secondaryPrizePool = gameInfo.secondaryPrizePool;
    uint256 reward;
    uint256[] memory rewards = new uint256[](agentIdsCount);

    for (uint256 i; i < agentIdsCount; ) {
        uint256 agentId = agentIds[i];
        _assertAgentOwnership(agentId);

        uint256 index = agentIndex(agentId);
        _assertAgentStatus(agents[index], agentId, AgentStatus.Active);

        uint256 totalEscapeValue = prizePool / currentRoundAgentsAlive;
        uint256 rewardForPlayer = (totalEscapeValue *
↪   _escapeMultiplier(currentRoundAgentsAlive)) /
            ONE_HUNDRED_PERCENT_IN_BASIS_POINTS;
        rewards[i] = rewardForPlayer;
        reward += rewardForPlayer;

        uint256 rewardToSecondaryPrizePool =
↪   (totalEscapeValue.unsafeSubtract(rewardForPlayer) *
            _escapeRewardSplitForSecondaryPrizePool(currentRoundAgentsAlive)) /
↪   ONE_HUNDRED_PERCENT_IN_BASIS_POINTS;

        unchecked {
            prizePool = prizePool - rewardForPlayer -
↪   rewardToSecondaryPrizePool;
        }
        secondaryPrizePool += rewardToSecondaryPrizePool;

        _swap({
            currentAgentIndex: index,
            lastAgentIndex: currentRoundAgentsAlive,
            agentId: agentId,
            newStatus: AgentStatus.Escaped
        });

        unchecked {
            --currentRoundAgentsAlive;
            ++i;
        }
    }

    // This is equivalent to
    // unchecked {
    //     gameInfo.activeAgents = uint16(activeAgentsAfterEscape);
```


```
    //     gameInfo.escapedAgents += uint16(agentIdsCount);
    // }
```

Threrefore, if the `heal` function is invoked first then the corresponding Wounded Agents will be healed in function `fulfillRandomWords`. If the `escape` function is invoked first and the number of `gameInfo.activeAgents` becomes equal to or less than `NUMBER_OF_SECONDARY_PRIZE_POOL_WINNERS`, the `heal` function will be disable. This obviously violates the fairness of the game.

Example

Consider the following situation:

After Round N, there are 100 agents alive. And, **1** Active Agent wants to `escape` and **10** Wounded Agents want to `heal`.

- Round N:
  - Active Agents: 51
  - Wounded Agents: 49
  - Healing Agents: 0

According to the order of execution, there are two situations. **Please note that the result is calculated only after `_healRequestFulfilled`, so therer are no new wounded or dead agents**

First, invoking `escape` before `heal`. `heal` is disable and all Wounded Agents are killed because there are not enough Active Agents.

- Round N+1:
  - Active Agents: 50
  - Wounded Agents: 0
  - Healing Agents: 0

Second, invoking `heal` before `escape`. Suppose that `heal` saves **5** agents, and we got:

- Round N+1:
  - Active Agents: 55
  - Wounded Agents: 39
  - Healing Agents: 0

Obviously, different execution orders lead to drastically different outcomes, which affects the fairness of the game.

Impact

If some Active Agents choose to escape, causing the count of `activeAgents` to become equal to or less than `NUMBER_OF_SECONDARY_PRIZE_POOL_WINNERS`, the Wounded Agents will lose their final chance to heal themselves.

This situation can significantly impact the game's fairness. The Wounded Agents would have otherwise had the opportunity to heal themselves and continue participating in the game. However, the escape of other agents leads to their immediate termination, depriving them of that chance.

Code Snippet

Heal will be disabled if there are not enout activeAgents. Infiltration.sol#L804

Escape will directly reduce the activeAgents. Infiltration.sol#L769

Tool used

Manual Review

Recommendation

It is advisable to ensure that the `escape` function is always called after the `heal` function in every round. This guarantees that every wounded agent has the opportunity to heal themselves when there are a sufficient number of `activeAgents` at the start of each round. This approach can enhance fairness and gameplay balance.

Discussion

0xhiroshi

This is a valid PvP game strategy.

nevillehuang

I can see sponsors view of disputing these issues given this protocol focuses on a PVP game. The difference between this two #43 and #57 and issue #98 is because #98 actually triggers an invalid game state and #84 truly skews the odds, but #43 and #57 don't fall into either categories.

However, due to a lack of concrete details on PVP strategies, I also see watsons point of view of how the use of `escape()` and `heal()` can cause unfair game mechanics. While I also understand the sponsor's view of difficulty in predicting PVP strategies, I think it could have been avoided by including this as potential risks in the accepted risks/known issues of sherlocks contest details (which is the source of truth for contest details).

As such, I am going to keep #43 and #57 as medium severity findings, given the attack path requires specific conditions that would otherwise have been valid PVP strategies.

0xhiroshi

We won't further dispute this issue and we won't fix it.

nevillehuang

After further considerations, going to mark this issue as invalid due to the analysis of the following test. It supports the sponsor claim that instant killing of wounded agents once active agents fall below `NUMBER_OF_SECONDARY_PRIZE_POOL_WINNERS` (50) is a valid PVP strategy.

1. It first simulates active agents falling to 51
2. Then it attempts to heal 3 wounded agents (ids: 8534, 3214 and 6189)
3. It then escapes 2 agents to make active agents fall below 49
4. A new round is started, instantly killing all wounded agents

Only #29 and #34 mentions the other root cause of instantly ending the game with `escape()` (which allows immediate claiming of grand prize) so this 2 issues will be dupped with #98.

Issue M-2: Wound agent can't invoke heal in the next round

Source: https://github.com/sherlock-audit/2023-10-looksrare-judging/issues/44

Found by

coffiasd, lil.eth, mstpr-brainbot, pontifex According to the document:

> if a user dies on round 12. The first round they can heal is round 13
> However incorrect current round id check led to users being unable to invoke the `heal` function in the next round.

Vulnerability Detail

Assume players being marked as wounded in the round `12` , players cannot invoke `heal` in the next round 13

```
function test_heal_in_next_round_v1() public {
    _startGameAndDrawOneRound();

    _drawXRounds(11);


    (uint256[] memory woundedAgentIds, ) = infiltration.getRoundInfo({roundId:
↪   12});

    address agentOwner = _ownerOf(woundedAgentIds[0]);
    looks.mint(agentOwner, HEAL_BASE_COST);

    vm.startPrank(agentOwner);
    _grantLooksApprovals();
    looks.approve(TRANSFER_MANAGER, HEAL_BASE_COST);

    uint256[] memory agentIds = new uint256[](1);
    agentIds[0] = woundedAgentIds[0];

    uint256[] memory costs = new uint256[](1);
    costs[0] = HEAL_BASE_COST;

    //get gameInfo
    (,,,,,uint40 currentRoundId,,,,,) = infiltration.gameInfo();
    assert(currentRoundId == 13);

    //get agent Info
    IInfiltration.Agent memory agentInfo =
↪   infiltration.getAgent(woundedAgentIds[0]);
    assert(agentInfo.woundedAt == 12);
```

```
    //agent can't invoke heal in the next round.
    vm.expectRevert(IInfiltration.HealingMustWaitAtLeastOneRound.selector);
    infiltration.heal(agentIds);
}
```

Impact

User have to wait for 1 more round which led to the odds for an Agent to heal successfully start at 99% at Round 1 reduce to 98% at Round 2.

Code Snippet

https://github.com/sherlock-audit/2023-10-looksrare/blob/main/contracts-infiltration/contracts/Infiltration.sol#L843#L847

```
// No need to check if the heal deadline has passed as the agent would be killed
unchecked {
    if (currentRoundId - woundedAt < 2) {
        revert HealingMustWaitAtLeastOneRound();
    }
}
```

Tool used

Manual Review

Recommendation

```
    // No need to check if the heal deadline has passed as the agent would be
↪   killed
    unchecked {
-       if (currentRoundId - woundedAt < 2) {
-       if (currentRoundId - woundedAt < 1) {
            revert HealingMustWaitAtLeastOneRound();
        }
    }
```

Discussion

nevillehuang

@0xhiroshi Any reason why you disagree with the severity? It does mentioned in the docs that the first round user can heal is right after the round his agent is

wounded. The above PoC also shows how a user cannot heal a wounded agent at round 13 when his agent was wounded at round 12.

0xhiroshi

https://github.com/LooksRare/contracts-infiltration/pull/151

SergeKireev

Fix LGTM

Issue M-3: Index values selected in `_woundRequestFulfilled()` are not uniformly distributed.

Source: https://github.com/sherlock-audit/2023-10-looksrare-judging/issues/84

Found by

0xGoodess, Tricko, cergyk, detectiveking Index values selected in `_woundRequestFulfilled()` are not uniformly distributed. Indexes right next to wounded agents are more likely to be selected in the subsequent iterations, leading to bias in the distribution of wounded agents.

Vulnerability Detail

At the end of each round, the function `_woundRequestFulfilled()` is called, which uses uses the `randomWord` obtained from the VRF to select which agents should be marked as wounded. This selection process is carried out by performing a modulo operation on the `randomWord` with respect to the number of agents currently alive in the round, and then adding 1 to the result. The resulting value corresponds to the index of the agent to be designated as wounded, as illustrated in the code snippet section.

However, if the resulting index corresponds to an agent who is already wounded, the `else` branch is executed, where 1 is added to the `randomWord` for the next iteration of the loop. **This is where the bias is introduced, because in the next iteration, the `woundedAgentIndex` will be the current `woundedAgentIndex` plus 1**. As can be seen below:

$$(A + 1) \bmod M$$

$$((A \bmod M) + (1 \bmod M)) \bmod M$$

As M > 1, we can simplify to

$$((A \bmod M) + 1) \bmod M$$

For $(A \bmod M) + 1$ less than M, we have

$$(A + 1) \bmod M = (A \bmod M) + 1$$

So with the exception of when `randomWord` overflows or (`randomWord` % `currentRoundAgentsAlive` + 1) >= `currentRoundAgentsAlive`, (`randomWord` + 1) %

`currentRoundAgentsAlive` will be equal to (`randomWord` % `currentRoundAgentsAlive`) + 1.

Consequently, when the `else` branch is triggered, the next `woundedAgentIndex` will be (previous `woundedAgentIndex`+ 1) from the last loop iteration (besides the two exceptions specified above). Therefore the agent at the next index will also be marked as wounded. As a result of this pattern, **agents whose indexes are immediately next to an already wounded agent are more likely to be wounded than the remaining agents**.

Consider the representative example below, albeit on a smaller scale (8 agents) to facilitate explanation. The initial state is represented in the table below:

() Index 1 2 3 4 5 6 7 8

()

Agents	Active	Active	Active	Active	Active	Active	Active	Active
Probabilities	0.125	0.125	0.125	0.125	0.125	0.125	0.125	0.125

()

In the initial iteration of `_woundRequestFulfilled()`, assume that index 2 is selected. As expected from function logic, for the next iteration a new `randomWord` will be generated, resulting in a new index within the range of 1 to 8, all with equal probabilities. However now not all agents have an equal likelihood of being wounded. This disparity arises because both 3 and 2 (due to the `else` branch in the code above) will lead to the agent at index 3 being wounded.

() Index 1 2 3 4 5 6 7 8

()

Agents	Active	Wounded	Active	Active	Active	Active	Active	Active
Probabilities	0.125	-	0.25	0.125	0.125	0.125	0.125	0.125

()

Now suppose that agents at index 2 and 3 are wounded. Following from the explanations above, index 4 has three times the chance of being wounded.

() Index 1 2 3 4 5 6 7 8

()

Agents	Active	Wounded	Wounded	Active	Active	Active	Active	Active

() Index 1 2 3 4 5 6 7 8

()

Probabilities 0.125 - - 0.375 0.125 0.125 0.125 0.125

()

Impact

The distribution of wounded status among indexes is not uniform. Indexes subsequent to agents already wounded are more likely to be selected, introducing unfairness in the distribution of wounded agents.

This is particularly problematic because indexes that are close to each other are more likely to owned by the same address, due to the fact that when minting multiple agents, they are created in sequential indexes. Consequently, if an address has one of its agents marked as wounded, it becomes more probable that additional agents owned by the same address will also be marked as wounded. This creates an unfair situation to users.

Code Snippet

https://github.com/sherlock-audit/2023-10-looksrare/blob/main/contracts-infiltration/contracts/Infiltration.sol#L1404-L1468

Tool used

Manual Review

Recommendation

Consider eliminating the `else` and calling `_nextRandomWord(randomWord)` at the end of the loop iteration, so a new `randomWord` is generated each time. As shown in the diff below:

```
diff --git a/Infiltration.sol b/Infiltration.mod.sol
index 31af961..1c43c31 100644
--- a/Infiltration.sol
+++ b/Infiltration.mod.sol
@@ -1451,15 +1451,9 @@ contract Infiltration is
                     ++i;
                     currentRoundWoundedAgentIds[i] = uint16(woundedAgentId);
                 }
-
-                randomWord = _nextRandomWord(randomWord);
-            } else {
```

```
-                 // If no agent is wounded using the current random word,
↪  increment by 1 and retry.
-                 // If overflow, it will wrap around to 0.
-                 unchecked {
-                     ++randomWord;
-                 }
              }
+
+             randomWord = _nextRandomWord(randomWord);
          }

          currentRoundWoundedAgentIds[0] = uint16(woundedAgentsCount);
```

Discussion

0xhiroshi

https://github.com/LooksRare/contracts-infiltration/pull/152

SergeKireev

Fix LGTM

Issue M-4: fulfillRandomWords() could revert under certain circumstances

Source: https://github.com/sherlock-audit/2023-10-looksrare-judging/issues/136

Found by

ge6a, klaus, p-tsanev

According the documentation of Chainlink VRF the max gas limit for the VRF coordinator is 2 500 000. This means that the max gas that fulfillRandomWords() can use is 2 500 000 and if it is exceeded the function would revert. I have constructed a proof of concept that demonstrates it is possible to have a scenario in which fulfillRandomWords reverts and thereby disrupts the protocol's work.

Vulnerability Detail

Crucial part of my POC is the variable AGENTS_TO_WOUND_PER_ROUND_IN_BASIS_POINTS. I communicated with the protocol's team that they plan to set it to 20 initially but it is possible to have a different value for it in future. For the POC i used 30.

```
function test_fulfillRandomWords_revert() public {
        _startGameAndDrawOneRound();

        _drawXRounds(48);

        uint256 counter = 0;
        uint256[] memory wa = new uint256[](30);
        uint256 totalCost = 0;

        for (uint256 j=2; j <= 6; j++)
        {
            (uint256[] memory woundedAgentIds, ) =
↪   infiltration.getRoundInfo({roundId: j});

            uint256[] memory costs = new uint256[](woundedAgentIds.length);
            for (uint256 i; i < woundedAgentIds.length; i++) {
                costs[i] = HEAL_BASE_COST;
                wa[counter] = woundedAgentIds[i];
                counter++;
                if(counter > 29) break;
            }

            if(counter > 29) break;
        }
```

```
        totalCost = HEAL_BASE_COST * wa.length;
        looks.mint(user1, totalCost);

        vm.startPrank(user1);
        _grantLooksApprovals();
        looks.approve(TRANSFER_MANAGER, totalCost);


        infiltration.heal(wa);
        vm.stopPrank();

        _drawXRounds(1);
    }
```

I put this test into Infiltration.startNewRound.t.sol and used --gas-report to see that the gas used for fulfillRandomWords exceeds 2 500 000.

Impact

DOS of the protocol and inability to continue the game.

Code Snippet

https://github.com/sherlock-audit/2023-10-looksrare/blob/main/contracts-infiltration/contracts/Infiltration.sol#L1096-L1249 https://docs.chain.link/vrf/v2/subscription/supported-networks/#ethereum-mainnet https://docs.chain.link/vrf/v2/security#fulfillrandomwords-must-not-revert

Tool used

Manual Review

Recommendation

A couple of ideas :

1) You can limit the value of AGENTS_TO_WOUND_PER_ROUND_IN_BASIS_POINTS to a small enough number so that it is 100% sure it will not reach the gas limit.
2) Consider simply storing the randomness and taking more complex follow-on actions in separate contract calls as stated in the "Security Considerations" section of the VRF's docs.

Discussion

nevillehuang

Accepted risks, the **KEYWORDS** here are:

- **periods of network congestion** --> this causes the hard code gas fallback to revert --> accepted risk
- **Any reason causing randomness request to not be fufilled** --> If request for randomness is not fufilled due to ANY reason, even if highilighted in a submission, it is not a accepted finding since it is an accepted risk LooksRare are willing to take

  In the case of extended periods of network congestion or any reason causing the randomness request not to be fulfilled, the contract owner is able to withdraw everything after approximately 36 hours.

gstoyanovbg

Escalate I disagree with the comment of @nevillehuang . Imagine a situation in which you start a game with 10,000 participants, go through many rounds, and at one point, the game stops and cannot continue. As far as I understand, the judge's claim is that the funds can be withdrawn after a certain time, which is true. However, will the gas fees be returned to all users who have healed agents or traded them on the open market in some way? And what about the time that the players have devoted to winning a game that suddenly stops due to a bug and cannot continue? Every player may have claims for significant missed benefits (and losses from gas fees). Now, imagine that this continues to happen again and again in some of the subsequent games. Will anyone even invest time and resources to play this game? My request to the judges is to review my report again because it has nothing to do with 'periods of network congestion' as stated and this issue is not listed under the "Accepted risks" section. Thanks. P.S During the contest i discussed this with the sponsor and confirmed that this is an issue (probably doesn't matter but wanted to mention it)

sherlock-admin2

> Escalate I disagree with the comment of @nevillehuang . Imagine a situation in which you start a game with 10,000 participants, go through many rounds, and at one point, the game stops and cannot continue. As far as I understand, the judge's claim is that the funds can be withdrawn after a certain time, which is true. However, will the gas fees be returned to all users who have healed agents or traded them on the open market in some way? And what about the time that the players have devoted to winning a game that suddenly stops due to a bug and cannot continue? Every player may have claims for significant missed benefits (and losses from gas fees). Now, imagine that this continues to happen again and again in some of the subsequent games. Will anyone even invest time

> and resources to play this game? My request to the judges is to review my report again because it has nothing to do with 'periods of network congestion' as stated and this issue is not listed under the "Accepted risks" section. Thanks. P.S During the contest i discussed this with the sponsor and confirmed that this is an issue (probably doesn't matter but wanted to mention it)

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

PlamenTSV

Escalate My issue 107 is the same and I agree with the above statement, as this is not an occurance that can randomly happen once, depending on potential parameter changes it can become extremely likely to happen every time. If every game has a high chance to get permanently dosed, even if emergency withdrawal is available, the game itself becomes unplayable. The accepted risks cover network congestion and the 36 period, but that period does not solve the insolvency, it just refunds some funds, not even everything that costed players and congestion is not the cause, but the badly gas optimized function.

sherlock-admin2

> Escalate My issue 107 is the same and I agree with the above statement, as this is not an occurance that can randomly happen once, depending on potential parameter changes it can become extremely likely to happen every time. If every game has a high chance to get permanently dosed, even if emergency withdrawal is available, the game itself becomes unplayable. The accepted risks cover network congestion and the 36 period, but that period does not solve the insolvency, it just refunds some funds, not even everything that costed players and congestion is not the cause, but the badly gas optimized function.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

nevillehuang

This is the only comment I will make about this submission, and I will leave the rest up to @nasri136 @Evert0x .

I agree this is a valid issue, but the fact is in the accepted risk section it mentions that ANY and i repeat ANY issue not just network congestion causing randomness

request to fail is an accepted risk. If the game is not completed, then admin can always call `emergencyWithdraw()`, so as long as this function is not DoSed (which is not possible unless game has ended), then I am simply following sherlocks rules:

> Hierarchy of truth: Contest README > Sherlock rules for valid issues > Historical decisions. While considering the validity of an issue in case of any conflict the sources of truth are prioritized in the above order. For example: In case of conflict between Sherlock rules vs Sherlock's historical decision, Sherlock criteria for issues must be considered the source of truth. In case of conflict between information in the README vs Sherlock rules, the README overrides Sherlock rules.

And yes private/discord messages does not matter:

> Discord messages or DM screenshots are not considered sources of truth while judging an issue/escalation especially if they are conflicting with the contest README.

gstoyanovbg

I understand your point. However, still believe that the 'Accepted Risks' section is not formulated well enough, and this case should definitely be excluded from its scope. I am aware of Sherlock's rules, but in my opinion, there are nuances even in them, and they cannot be 100% accurate for all situations. I see that report #107 by @PlamenTSV, which is same issue as this, has labels "Sponsor Confirmed" and "Will fix." Also, @0xhiroshi has made a pull request for a fix which makes me happy because at the end of the day our goal is to have a secure, bug free protocol. However, it sounds strange to claim that a vulnerability is an "accepted risk" but at the same time to fix it. Looking forward for the final decision of the judges. If you have any questions, I am ready to assist.

nevillehuang

@gstoyanovbg I totally get your point too and agree with you. If I didn't have to follow sherlock rules, I would have side with the watsons and validated this finding. But the fact that in the accepted risks section they used the word ANY is too strong of a word for me to ignore.

PlamenTSV

Don't sherlock rules serve as a guideline? I hope most if not every watson that has read this issue agrees it is valid. The judges agree it is valid by severity and even the sponsors themselves want to fix it and deem it a valid finding. I think the reason the README is like this is because the protocol team did not expect that their protocol could have a DoS of this caliber - thus why they overlook the README and confirm the issue. It would be a shame to get robbed of a valid finding with no reward, worse ratio for the platform, but for the protocol to aknowledge and fix. There should be some kind of workaround for these kinds of scenarios. I believe that would be most fair and I hope you agree. If we could get a sherlock admin to

give clarity to the situation it would be appreciated, thanks to everyone in the comments.

PlamenTSV

> This is the only comment I will make about this submission, and I will leave the rest up to @nasri136 @Evert0x .
>
> I agree this is a valid issue, but the fact is in the accepted risk section it mentions that ANY and i repeat ANY issue not just network congestion causing randomness request to fail is an accepted risk. If the game is not completed, then admin can always call `emergencyWithdraw()`, so as long as this function is not DoSed (which is not possible unless game has ended), then I am simply following sherlocks rules:
>
> > Hierarchy of truth: Contest README > Sherlock rules for valid issues > Historical decisions. While considering the validity of an issue in case of any conflict the sources of truth are prioritized in the above order. For example: In case of conflict between Sherlock rules vs Sherlock's historical decision, Sherlock criteria for issues must be considered the source of truth. In case of conflict between information in the README vs Sherlock rules, the README overrides Sherlock rules.
>
> And yes private/discord messages does not matter:
>
> > Discord messages or DM screenshots are not considered sources of truth while judging an issue/escalation especially if they are conflicting with the contest README.

One thing to add while rereading this comment, shouldn't Sponsor final confirmation »> contest readmi, exactly because some issue's severity can go overlooked, just like in the current case

nevillehuang

Afaik, Sponsor confirmation has never been the deciding factor for a finding. It has always been sherlock rules, contest details and the code itself. To note, this rules are introduce to ensure judging consistency in the first place.

I agree that this finding is valid, and it is unfair to watsons for not being rewarded. But it is also very unfair to me for potentially having my judging points penalized because the source of ambiguity is not because of me. I think all watsons know judging is not easy, and the rules revolving judging is extremely strict. Even one miss finding by me will heavily penalize me.

I leave this in the hands of @Evert0x @nasri136 and will respect any final outcome.

Czar102

First off, the status of whether the sponsor wants to fix the issue has nothing to do with its validity. Please don't use it as an argument of judgement.

Secondly, I understand @nevillehuang about the strictness of the judging rules and potential payout issues. Will see what can I do about it in this case. The mechanism shouldn't bias you towards maintaining the current judgement, so if you have concerns with that, we can try to resolve those rewarding formula issues in DMs.

About the issue, it is true that it is unfortunately conflicting with the README. I think we should rethink the way the docs describes the hierarchy of truth.

Audits exist not only to show outright mistakes in the code. Auditors' role is to show that some properties (that are important to parties using the code) are not enforced or fulfilled. They need to think in a broader sense than the sponsors did, in order to provide any more insight above the technical one. Auditors need to teach sponsors not only about issues, but also about the impacts. "Why is this property a problem?"

Sponsors can't know what wording to use not to exclude the bugs they don't care about. Because that's not their job. It's our job to understand their needs. We work for them to secure their code. We tell them what do they need.

What sponsors probably meant by "any reason causing the randomness request not to be fulfilled" is most likely "any **external** reason causing the randomness request not to be fulfilled". They didn't think it could be their contract causing this issue. I think some Watsons correctly identified what sponsors intended to convey and should be rewarded for submitting the issue.

Will alter the docs to account for this kind of situations.

Could you share your feedback on the above approach? @nevillehuang @PlamenTSV @gstoyanovbg

PlamenTSV

I am glad you understand the aspect of the README file overlooking the potential issues. It is not that they do not accept issues like this, it is the fact that they did not expect their contract to be at fault for them. I believe us watsons did in fact identify a valid problem, adhering both to the sherlock criteria and the sponsor's needs (we understand their confirmation is not a valid judgement, but it can be an argument that they did a mistake in the README), and hopefully I am speaking for everyone when I say I am glad we reached such a fair outcome. I am looking forward to seeing some docs changes for edge-case scenarios like this so they can be more easily resolved in the future. Thanks for the hard work!

nevillehuang

Hi @Czar102, I will share with you how I interpreted this while judging. In addition to the conflicted READ.ME, the `emergencyWithdraw` function exists in the first place to deal with such scenarios, that is why i deemed it as an accepted risk since funds can always be rescued by trusted admins.

While I also understand the other side of the equation, I will share with you what I think is the only possible fair outcome for both watsons auditing and judging. I also want to clarify that unfairness goes both ways, but arguably even more so for judging since judging has way stricter rules.

> Hi all heres my input for the findings regarding 107 & 136. In my opinion, the only fair outcome for all parties for 107 & 136 is to validate the finding as medium severity and waive its potential impact on my judging status. After all, the finding has provided significant value to the sponsor given the fix implemented. But you can see from the current judging state that there will likely be less than 10 issues, so one missed issue by me can lead to heavy penalisation of my judging points or maybe even lead me to not make the minimum 80% recall threshold. I have included this in part of my growing list of suggestion for improvement to sherlock, that is we possibly need a period of 24-48 hour for sponsor to clarify/edit any doubts regarding contest details. Lead judge/head of judging can facilitate in this. Of course this is just my opinion, I leave the rest up to the temporary head of judging and sherlock

gstoyanovbg

@Czar102, I agree with everything you've said, and I share your philosophy on the work of auditors. Regarding the rules for judging, in my opinion, they cannot cover all possible cases, especially those that have not arisen before. They should evolve over time, and in the event of a situation, the Sherlock team should be able to make a decision so that there are no affected judges and auditors. @nevillehuang, I still believe this is a high severity issue, but at the same time, it is not fair to be punished because you acted according to the defined rules. I hope a solution can be found.

Czar102

> the `emergencyWithdraw` function exists in the first place to deal with such scenarios

Nevertheless, the contract doesn't function correctly. I think that because it is possible to rescue funds, it can be a medium severity issue. Players who were supposed to win in a game don't get their funds, but the EV doesn't change. The result is never uncovered. Hence medium.

Because judges who approached this issue "by the book" would lose out on this outcome, this issue and duplicates will not count towards the judging contest reward calculation.

@nevillehuang @nasri136 Please let me know whether, according to the above approach, #72 should be considered valid or not. It seems it presents another issue.

Czar102

Result: Medium Has duplicates

The issue and duplicates will not be counted towards the judging payout.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- gstoyanovbg: accepted
- PlamenTSV: accepted

gstoyanovbg

@Czar102, allow me to disagree with the severity of the report. I'm not sure if I have the right to dispute it after the escalation is resolved, but here are my arguments:

1) I agree that the funds can be rescued using emergencyWithdraw(). However, the funds locked in one iteration of the game are insignificant compared to the indirect losses from such an event. The success of luck-based games is directly tied to the trust of the players in the game creators. If the game is interrupted at a crucial moment (which is very likely), the affected players will certainly question the fairness of the game. Can they be sure that this is not intentionally caused for someone not to win the prize? The damage to the brand's reputation will be irreversible. In addition to the missed profits from hundreds of future iterations of the game, developers will incur losses from the funds invested in development.
2) I mentioned earlier that when we talk about financial losses, we should also consider the losses for users from gas fees (may have thousands of players). We all know what the fees on the Ethereum mainnet can be, especially in moments of network congestion. For a player whose agent is wounded, it may be extremely important to heal it regardless of the fee paid. When funds are withdrawn through emergencyWithdraw() and returned to the players, they should be compensated for gas fees. These funds must be paid either by Looksrare or the players. In both cases, someone loses.
3) The time lost by players to play a game that suddenly stops and cannot continue should also be taken into account. Even if the game starts again from the beginning and everything is fine, it cannot start from the same state it was interrupted. A player may have been in a position where they were likely to win a prize, but they probably won't be compensated for it. Even if they are, it will be at the protocol's expense.

Considering 1), 2), and 3), my position is that there are much larger losses for each party than just the funds locked in the contract at the time of the interruption.

Czar102

> Can they be sure that this is not intentionally caused for someone not to win the prize?

If that's the case, this could have been a high severity issue. I don't think that the report mentions such a scenario though.

> The damage to the brand's reputation will be irreversible.

That's true, but that is not a high severity vulnerability. A high severity vulnerability is when there is a direct and very probable loss of funds, which is not the case here.

> These funds must be paid either by Looksrare or the players. In both cases, someone loses.

The gas costs are out of scope here. The fact that the user plays the game is a "value gotten" for the gas. Anyway, even if, there would be no high severity impact because of gas.

> A player may have been in a position where they were likely to win a prize, but they probably won't be compensated for it.

We don't know that. Maybe the protocol would distribute the rewards proportionally to the EV of the players in a given position given an optimal strategy?

This is why I chose a Medium severity for this issue.

gstoyanovbg

> If that's the case, this could have been a high severity issue. I don't think that the report mentions such a scenario though.

Me and you know that this is not true, but for people which are not solidity developers / auditors it is not clear and creates reasonable doubt about the fairness of the game and its creators

> That's true, but that is not a high severity vulnerability. A high severity vulnerability is when there is a direct and very probable loss of funds, which is not the case here.

I agree there is no direct loss of funds but we have an issue that breaks core contract functionality, rendering the protocol/contract useless + indirect loss of funds on a large scale.

> We don't know that. Maybe the protocol would distribute the rewards proportionally to the EV of the players in a given position given an optimal strategy?

That would be the right decision. However, there is no way to do it in a good enough manner because there is no way to prove which player had what financial resources for the game. This is a very important variable for the potential mathematical model. For example, two agents who have survived until round X and have been healed three times will have equal weight if the game is interrupted at that moment. However, the player behind the first may no longer have the financial ability to heal the agent, while the second may be able to heal it three more times without any problem.

Czar102

> Me and you know that this is not true, but for people which are not solidity developers / auditors it is not clear and creates reasonable doubt about the fairness of the game and its creators

This is exactly why we have this job. We need to tell them. And this impact doesn't seem to be the case here.

> However, there is no way to do it in a good enough manner because there is no way to prove which player had what financial resources for the game.

Yes, but I feel that's an insufficient impact to consider a high severity impact. In the end, they might make another contract and continue the game ;) Ofc that would be costly but the loss of funds is extremely limited here. This is why it's Medium.

0xhiroshi

https://github.com/LooksRare/contracts-infiltration/pull/165

SergeKireev

Fix LGTM