Primodium Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, Shaka, ubermensch

October 2nd - October 8th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

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

3. Introduction

A time-boxed security review of the **primodiumxyz/primodium-empires-audit-pashov** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Primodium

The protocol rewards players based on their points in the winning empire, distributing funds for withdrawal, and allows players to influence gameplay by locking points. Prices adjust dynamically with each transaction, and a pseudorandom function selects the winner in tie scenarios.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 10ee58e648e0d457cec789443a9589dd83b6b307

fixes review commit hash - f757766f191be09b9f842ae1345a573b12392b22

Scope

The following smart contracts were in scope of the audit:

- `PointsMap`
- `InitPrice`
- `LibOverride`
- `LibPoints`
- `LibPrice`
- `WithdrawRakeSystem`
- `RewardsSystem`
- `ResetSystem`
- `OverrideShipSystem`
- `OverrideShieldSystem`
- `OverrideShieldEaterSystem`
- `OverridePointsSystem`
- `OverrideMagnetsSystem`
- `OverrideAirdropSystem`
- `OverrideAcidSystem`
- `EmpiresSystem`
- `AdminSystem`
- `ResetClearLoopSubsystem`
- `PayoutManager.sol`

7. Executive Summary

Over the course of the security review, 0xunforgiven, Shaka, ubermensch engaged with Primodium to review Primodium. In this period of time a total of **23** issues were uncovered.

Protocol Summary

Protocol Name	Primodium
Repository	https://github.com/primodiumxyz/primodium-empires-audit-pashov
Date	October 2nd - October 8th
Protocol Type	Game

Findings Count

Severity	**Amount**
Critical	4
Medium	10
Low	9
Total Findings	**23**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Players might not be able to withdraw earnings if the game is reset	Critical	Resolved
[C-02]	The portion of the pot corresponding to locked points is not distributed to the players	Critical	Resolved
[C-03]	MagnetTurnPlanets table is not cleared when the game is reset	Critical	Resolved
[C-04]	Lack of access control in ResetSystem::resetGame()	Critical	Resolved
[M-01]	Asymmetry in the cost and points of overrides	Medium	Resolved
[M-02]	Purchase of overrides can increase a player's loss more than the actual amount spent	Medium	Resolved
[M-03]	Modifier onlyNotGameOver() does not check for Domination Victory	Medium	Resolved
[M-04]	Sandwich attack for turnEmpirePointPriceDown()	Medium	Acknowledged
[M-05]	Victors list can be big and cannot be handled in one transaction	Medium	Resolved
[M-06]	Plant's empire can change by front-running in purchase transactions	Medium	Resolved
[M-07]	clearLoop() can OOG because of the nested loop over empires and planets	Medium	Resolved
[M-08]	OOG error in clearLoop()	Medium	Acknowledged

[M-09]	Vulnerability to MEV sandwich attacks in point purchase and sale system	Medium	Resolved
[M-10]	placeMagnet does not check if the planet is owned by the empire	Medium	Acknowledged
[L-01]	Incorrect game over check in _checkTimeVictory function	Low	Resolved
[L-02]	Wrong value assigned to the ethSpent field of the PlaceAcidOverrideLog table	Low	Resolved
[L-03]	PostManager.record() allows skipping rounds	Low	Resolved
[L-04]	Game uses predictable randomness to select a winner	Low	Acknowledged
[L-05]	There are two rewards claiming systems that are not compatible	Low	Acknowledged
[L-06]	Shield Eater can be detonated on unowned planets	Low	Acknowledged
[L-07]	In delegated calls excess ether is refunded to the delegator instead of the caller	Low	Acknowledged
[L-08]	Users cannot sell their points in some scenarios	Low	Acknowledged
[L-09]	Attacker can perform reentrancy and cause harm for off-chain logic	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Players might not be able to withdraw earnings if the game is reset

Severity

Impact: High

Likelihood: High

Description

The funds of the pot are meant to be sent to the `PayoutManager` contract so that players can withdraw their earnings at their leisure. However, in the current implementation, the funds are not sent to the `PayoutManager` contract, so the withdrawal of earnings is dealt with by the `RewardsSystem.withdrawEarnings` function.

The issue is that the `RewardsSystem.withdrawEarnings` function can only be called if the game is over, meaning that players can only withdraw their earnings for the time between the end of the game and the beginning of the next game. If the game is reset, the players will not be able to withdraw their earnings.

Note that there is another reported issue about the lack of checks in the `resetGame` function, which allows the game to be reset at any time by anyone. But even after fixing that issue, the players will still not be able to withdraw their earnings if the game is reset.

Recommendations

Replace the `RewardsSystem` for a new system that ends the game when the required conditions are met, and sends the funds to the `PayoutManager`

contract.

[C-02] The portion of the pot corresponding to locked points is not distributed to the players

Severity

Impact: High

Likelihood: High

Description

The `withdrawEarnings` function in the `RewardsSystem` contract calculates the amount of pot to distribute to the player based on the number of points they have in the winning empire. These are some of the steps in the function:

1. Get the total points issued to the winning empire (line 131).
2. Get the player's points in the winning empire (line 136), excluding the locked points (line 137).
3. Calculate the player's share of the pot based on their points (line 142).

```
File: RewardsSystem.sol

125:   function withdrawEarnings() public _onlyGameOver {
126:     EEmpire winningEmpire = WinningEmpire.get();
127:     require
  (winningEmpire != EEmpire.NULL, "[RewardsSystem] No empire has won the game");
128:
129:     bytes32 playerId = addressToId(_msgSender());
130:
131:     uint256 empirePoints = Empire.getPointsIssued(winningEmpire);
132:     if (empirePoints == 0) {
133:       return;
134:     }
135:
136:     uint256 playerEmpirePoints = PointsMap.getValue
  (winningEmpire, playerId) -
137:       PointsMap.getLockedPoints(winningEmpire, playerId);
138:
139:     if (playerEmpirePoints == 0) return;
140:
141:     uint256 pot = (Balances.get(EMPIRES_NAMESPACE_ID));
142:     uint256 playerPot = (pot * playerEmpirePoints) / empirePoints;
143:
144:     PlayersMap.setGain(playerId, PlayersMap.get
  (playerId).gain + playerPot);
145:     PointsMap.remove(winningEmpire, playerId);
146:
147:     IWorld(_world()).transferBalanceToAddress
  (EMPIRES_NAMESPACE_ID, _msgSender(), playerPot);
148:   }
```

The issue is that `empirePoints` includes both locked and unlocked points, while `playerEmpirePoints` only includes unlocked points. This means that a portion of the pot corresponding to locked points is not distributed to the players and is locked in the contract.

Let's consider the following scenario:

- The total points issued to the winning empire is 15 and the total pot is 15 ether.
- Player A has 10 points in the winning empire, 5 of which are locked.
- Player B has 5 points in the winning empire, all unlocked.
- Player A withdraws 5 ether from the pot: `15 ether * (10 points - 5 points) / 15 points`.
- Player B withdraws 5 ether from the pot: `15 ether * 5 points / 15 points`.
- The remaining 5 ether in the pot is locked in the contract.

Recommendations

Option 1: Unlock all points for the winning empire when the game ends. This way, the portion of the pot corresponding to locked points is distributed to the

players that locked them based on their number of locked points.

Option 2: Subtract the total points locked in the winning empire from `empirePoints`. This way, the portion of the pot corresponding to locked points is distributed evenly among all players based on their number of unlocked points.

[C-03] `MagnetTurnPlanets` table is not cleared when the game is reset

Severity

Impact: High

Likelihood: High

Description

The `resetGame` function in the `ResetSystem` contract resets the game by clearing the necessary tables and setting the initial data.

However, the `MagnetTurnPlanets` table is not cleared when the game is reset. The data of this table is used every time `updateWorld` is called to check what magnets should be removed.

```
File: UpdateMagnetsSubsystem.sol

contract UpdateMagnetsSubsystem is EmpiresSystem {
  function updateMagnets() public {
    uint256 nextTurn = Turn.getValue() + 1;
    // remove magnets that should be removed for the next turn for each empire,
    // so the turn starts on an updated state
    for (uint8 i = 1; i <= P_GameConfig.getEmpireCount(); i++) {
      EEmpire empire = EEmpire(i);
@>    bytes32[] memory magnetEmpireTurnPlanets = MagnetTurnPlanets.get
  (empire, nextTurn);
      for (uint j = 0; j < magnetEmpireTurnPlanets.length; j++) {
        // clear magnet
@>      LibMagnet.removeMagnet(empire, magnetEmpireTurnPlanets[j]);
      }

      MagnetTurnPlanets.deleteRecord(empire, nextTurn);
    }
  }
}
```

Leaving this data in the table will provoke that magnets might be removed before they should be, breaking the game's mechanics and causing users who place magnets to pay for more turns than they should.

See Proof of concept for a detailed example.

Proof of concept:

Add the following contract to the `test` directory and run `pnpm mud test --forgeOptions='--mt test_magnet -vv' --skipBuild`.

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.24;

import { console, PrimodiumTest } from "test/PrimodiumTest.t.sol";
import
   { P_GameConfig, Planet, Turn, Magnet, MagnetData, TurnData, MagnetTurnPlanets } fro
import { EEmpire, EOverride } from "codegen/common.sol";
import { LibPrice } from "libraries/LibPrice.sol";
import { RoutineThresholds } from "src/Types.sol";
import { PlanetsSet } from "adts/PlanetsSet.sol";
import { CitadelPlanetsSet } from "adts/CitadelPlanetsSet.sol";

contract AuditTest is PrimodiumTest {
  uint256 constant ROUND_TURNS = 500;
  uint256 constant TURN_LENGTH_BLOCKS = 3;
  uint256 internal EMPIRE_COUNT;
  uint256 internal GAME_OVER_BLOCK;
  bytes32 internal planetId;

  function setUp() public override {
    super.setUp();

    EMPIRE_COUNT = P_GameConfig.getEmpireCount();
    GAME_OVER_BLOCK = block.number + ROUND_TURNS * TURN_LENGTH_BLOCKS;
    planetId = _getFirstPlanet(EEmpire(1));

    vm.startPrank(creator);
    P_GameConfig.setTurnLengthBlocks(TURN_LENGTH_BLOCKS);
    P_GameConfig.setGameOverBlock(GAME_OVER_BLOCK);
    Turn.setNextTurnBlock(block.number + TURN_LENGTH_BLOCKS);
    vm.stopPrank();
  }

  function test_magnetTurnPlanetsNotCleared() public {
    // EEmpire.RED conquers all citadels
    createShips(95);
    skipTurns(EMPIRE_COUNT);
    for (uint256 i = 0; i < EMPIRE_COUNT; i++) {
      vm.roll(block.number + TURN_LENGTH_BLOCKS);
      createPendingMove();
      skipTurns(2 * EMPIRE_COUNT - 1);
      createShips(95);
    }

    // Alice places a magnet that is meant to be removed after 1 full turn
    placeMagnet(1, alice);
    uint256 aliceMagnetDeletionTurn = Turn.getValue() + 1 * EMPIRE_COUNT;
    bytes32[] memory magnetEmpireTurnPlanets = MagnetTurnPlanets.get(EEmpire
      (1), aliceMagnetDeletionTurn);
    assert(magnetEmpireTurnPlanets.length == 1);

    // Game ends by domination
    vm.prank(alice);
    world.Empires__withdrawEarnings();

    // Game is reset, but MagnetTurnPlanets is not cleared
    world.Empires__resetGame();
    magnetEmpireTurnPlanets = MagnetTurnPlanets.get(EEmpire
      (1), aliceMagnetDeletionTurn);
    assert(magnetEmpireTurnPlanets.length == 1);

    // Game advances until aliceMagnetDeletionTurn - 1
    skipTurns(aliceMagnetDeletionTurn - 2);

    // Bob places a magnet for 10 full turns (10 * EMPIRE_COUNT)
    placeMagnet(10, bob);
    MagnetData memory magnetData = Magnet.get(EEmpire(1), planetId);
```

```
    uint256 bobMagnetDeletionTurn = Turn.getValue() + 10 * EMPIRE_COUNT;
    assert(magnetData.endTurn == bobMagnetDeletionTurn);

    // One turn passes, reaching aliceMagnetDeletionTurn, and the magnet Bob
    // placed for 10 turns is removed
    skipTurns(1);
    magnetData = Magnet.get(EEmpire(1), planetId);
    assert(!magnetData.isMagnet);

    // The data for Bob's magnet deletion is still present, so if Charlie places
    // a magnet,
    // the same issue will occur, and so on
    magnetEmpireTurnPlanets = MagnetTurnPlanets.get(EEmpire
      (1), bobMagnetDeletionTurn);
    assert(magnetEmpireTurnPlanets.length == 1);
  }

  function skipTurns(uint256 turns) public {
    for (uint256 i = 0; i < turns; i++) {
      vm.roll(block.number + TURN_LENGTH_BLOCKS);
      if (block.number < GAME_OVER_BLOCK) {
        executePendingMove();
      }
    }
  }

  function createShips(uint256 numShips) public {
    uint256 cost = LibPrice.getTotalCost(EOverride.CreateShip, EEmpire
      (1), numShips);
    vm.prank(alice);
    world.Empires__createShip{ value: cost }(planetId, numShips);
  }

  function placeMagnet(uint256 fullTurnDuration, address player) public {
    uint256 cost = LibPrice.getTotalCost(EOverride.PlaceMagnet, EEmpire
      (1), fullTurnDuration);
    vm.prank(player);
    world.Empires__placeMagnet{ value: cost }(EEmpire
      (1), planetId, fullTurnDuration);
  }

  function createPendingMove() public {
    RoutineThresholds[] memory routineThresholds = new RoutineThresholds[](1);
    routineThresholds[0] = RoutineThresholds({
      planetId: planetId,
      moveTargetId: _getNotOwnedCitadel(Turn.get().empire),
      accumulateGold: 0,
      buyShields: 0,
      buyShips: 0,
      moveShips: 10000
    });

    vm.prank(creator);
    world.Empires__updateWorld(routineThresholds);
  }

  function executePendingMove() public {
    RoutineThresholds[] memory routineThresholds = new RoutineThresholds[](1);
    routineThresholds[0] = RoutineThresholds({
      planetId: planetId,
      moveTargetId: bytes32(0),
      accumulateGold: 10000,
      buyShields: 0,
      buyShips: 0,
      moveShips: 0
    });

    vm.prank(creator);
```

```
    world.Empires__updateWorld(routineThresholds);
  }

  function _getNotOwnedCitadel(EEmpire empire) public view returns
    (bytes32 planetId_) {
    bytes32[] memory citadelPlanets = CitadelPlanetsSet.getCitadelPlanetIds();
    for (uint256 i = 0; i < citadelPlanets.length; i++) {
      if (empire != Planet.getEmpireId(citadelPlanets[i])) {
        return citadelPlanets[i];
      }
    }
  }

  function _getFirstPlanet(EEmpire empire) internal view returns
    (bytes32 planetId_) {
    bytes32[] memory allPlanets = PlanetsSet.getPlanetIds();
    for (uint256 i = 0; i < allPlanets.length; i++) {
      if (Planet.getEmpireId(allPlanets[i]) == empire) {
        return allPlanets[i];
      }
    }
  }
}
```

Recommendations

Clear the `MagnetTurnPlanets` table when the game is reset.

[C-04] Lack of access control in `ResetSystem::resetGame()`

Severity

Impact: High

Likelihood: High

Description

The `resetGame` function in the `ResetSystem` contract does not implement any form of access control, allowing anyone to call it. This creates a severe vulnerability, as any unauthorized user can reset the entire game state, leading to potential loss of progress and disruption of gameplay.

Additionally, in the configuration file `mud.config.ts`, the `openAccess` flag for several subsystems, including `ResetClearLoopSubsystem`, is correctly set to `false`, but this access control has not been applied to the `ResetSystem` contract. The absence of a modifier enforcing access restrictions exacerbates the issue.

Here is the code where access control is missing:

```
function resetGame() public {
    IWorld world = IWorld(_world());
    world.Empires__clearLoop();
    P_GameConfigData memory config = P_GameConfig.get();

    P_GameConfig.setGameOverBlock
      (block.number + config.nextGameLengthTurns * config.turnLengthBlocks);
    P_GameConfig.setGameStartTimestamp(block.timestamp);
    createPlanets(); // Planet and Empire tables are reset to default values
    LibShieldEater.initialize
    //(); // ShieldEater relocated, charge reset, and destination set
    initPrice
    //(); // Empire.setPointPrice and OverrideCost tables are reset to default values
    Turn.set(block.number + config.turnLengthBlocks, EEmpire.Red, 1);
}
```

This lack of control could lead to a malicious actor resetting the game at any time, which could have disastrous consequences, especially in a competitive or high-stakes environment.

Recommendations

- **Implement access control modifiers**: Add a restrictive modifier such as `onlyOwner` or `onlyAdmin` to ensure that only authorized users can call the `resetGame` function.
- **Update configuration**: Ensure that `openAccess` in the `mud.config.ts` file is correctly applied and that access control is enforced at the contract level as well.

```
export const worldInput = {
  namespace: "Empires",
  systems: {
    UpdateEmpiresSubsystem: { openAccess: false },
    UpdateAcidSubsystem: { openAccess: false },
    UpdateMagnetsSubsystem: { openAccess: false },
    UpdatePriceSubsystem: { openAccess: false },
    UpdateShieldEaterSubsystem: { openAccess: false },
    ResetClearLoopSubsystem: { openAccess: false },
+   ResetSystem: { openAccess: false },
  },
```

8.2. Medium Findings

[M-01] Asymmetry in the cost and points of overrides

Severity

Impact: Low

Likelihood: High

Description

The cost of overrides and the number of points received from purchasing them are dependent on the number of empires in the game. However, for regress overrides the defeated empires are not taken into account, while for progress overrides all empires are considered. This creates an asymmetry in the cost and points received for the two types of overrides.

If for example there are only two undefeated empires left in the game, and assuming constant point prices, the cost of a regress override will only be 5 times lower than the cost at the beginning of the game, while the cost of a progress override will not change at all.

Recommendations

```
function getProgressPointCost(
      EOverride_overrideType,
      EEmpire_empireImpacted,
      uint256_overrideCount
    ) internal view returns (uint256
        uint8 empireCount = P_GameConfig.getEmpireCount();
+       uint8 enemiesCount;
+       for (uint8 i = 1; i <= empireCount; i++) {
+           if (EEmpire(i) != _empireImpacted && !Empire.getIsDefeated(EEmpire
+ (i))) {
+               enemiesCount++;
+           }
+       }
+
-       return getPointCost(_empireImpacted, _overrideCount *
- (empireCount - 1) * P_PointConfig.getPointUnit() * P_OverrideConfig.getPointMultipli
+       return getPointCost
+ (_empireImpacted, _overrideCount * enemiesCount * P_PointConfig.getPointUnit() * P_O
    }
```

```
function _purchaseOverride(
(...)
    if (progressOverride) {
+       uint8 enemiesCount;
+       for (uint8 i = 1; i <= empireCount; i++) {
+           if (EEmpire(i) != _empireImpacted && !Empire.getIsDefeated(EEmpire
+ (i))) {
+               enemiesCount++;
+           }
+       }
-       uint256 numPoints = _overrideCount *
- (empireCount - 1) * pointUnit * pointMultiplier;
+
+        uint256 numPoints = _overrideCount * enemiesCount * pointUnit * pointMultipli
```

[M-02] Purchase of overrides can increase a player's loss more than the actual amount spent

Severity

Impact: Low

Likelihood: High

Description

The override systems call the `LibOverride._purchaseOverride` function, which handles the purchase of the respective override. The function receives the `_spend` parameter, which is the amount spent on the override, and this value is used to increase the `loss` of the player.

```
File: LibOverride.sol

36    PlayersMap.setLoss(playerId, PlayersMap.get(playerId).loss + _spend);
```

However, the value passed to the `_spend` parameter is `_msgValue()`, which can be higher than the actual amount spent by the player, as the excess value sent is refunded to the player.

```
File: OverrideAcidSystem.sol

32    LibOverride._purchaseOverride
  (playerId, EOverride.PlaceAcid, empire, 1, _msgValue());
```

As a result, the player's `loss` will be higher than the actual amount spent on the override.

Recommendations

For all override systems functions, pass the `cost` of the override to the `_purchaseOverride` function instead of `_msgValue()`.

[M-03] Modifier `onlyNotGameOver()` does not check for Domination Victory

Severity

Impact: Medium

Likelihood: Medium

Description

Some functions have the `onlyNotGameOver()` modifiers like selling points or buying overrides. `onlyNotGameOver()` checks the game deadline and winner to check for game over:

```
modifier _onlyNotGameOver() {
    require(Ready.get(), "[EmpiresSystem] Game not ready");
    require(WinningEmpire.get() == EEmpire.NULL, "[EmpiresSystem] Game over");
    uint256 endBlock = P_GameConfig.getGameOverBlock();
    require
      (endBlock == 0 || block.number < endBlock, "[EmpiresSystem] Game over");
    _;
  }
```

The issue is that the code doesn't check for domination victory scenarios. So the game could be over but this modifier won't detect it. This issue would allow users to call those protected functions and perform actions while the game is practically over. One important scenario is when users see that the game is about to finish (with domination scenario) they can front-run transaction that is about to set value for `WinningEmpire` and sell their points in losing empires while `WinningEmpire` is `NULL`.

Recommendations

Call `_checkDominationVictory()` and make sure the game isn't over because of the domination scenario.

[M-04] Sandwich attack for `turnEmpirePointPriceDown()`

Severity

Impact: Medium

Likelihood: Medium

Description

Admins call `turnEmpirePointPriceDown()` and reduce the price of an empire's points:

```
function turnEmpirePointPriceDown(EEmpire _empire) internal {
    P_PointConfigData memory config = P_PointConfig.get();
    uint256 newPointPrice = Empire.getPointPrice(_empire);
    if (newPointPrice >= config.minPointPrice + config.pointGenRate) {
      newPointPrice -= config.pointGenRate;
    } else {
      newPointPrice = config.minPointPrice;
    }
    Empire.setPointPrice(_empire, newPointPrice);
    HistoricalPointPrice.set(_empire, block.timestamp, newPointPrice);
  }
```

The issue is that the code subtracts an absolute value from the empire's point's price. So if the price of the points was lower then the percentage price decrease would be higher and users can use this to profit from price decrease by sandwich attacks. Users can't buy points directly and they need to buy override but the result is the same. Users can time their override buys with the admin's transaction and perform the sandwich attack to benefit from the admin's transactions. This is the POC: (buying points means buying some override which results in buying points)

1. Suppose an empire's point's price is 100.
2. User1 has 50 points in that empire and if he buys one more token it would cost 100.
3. Admins call `turnEmpirePointPriceDown()` to reduce the price by 10 units.

4. User1 would perform a sandwich attack for the admin's transaction and first he would sell his 50 points so the price would be decreased to 50.
5. Then the admin's transaction would be executed and the point's price would be 49.
6. Now User1 would buy 1 token with the price of 50 units and buy the sold 50 points with their sell price.
7. In the end user was able to buy 1 token with a price of 50 instead of 100.

Recommendations

Use percentage or value(`point * price`) based on price decrease or decrease the price over time.

[M-05] Victors list can be big and cannot be handled in one transaction

Severity

Impact: Medium

Likelihood: Medium

Description

Contract PayoutManager has been used to distribute the winner's payout. Admins call `record()` to set winners and their reward amounts and code loops through those winners and sets their rewards in the storage.

```
function record(
        address[] memory _victors,
        uint256[] memory _gains,
        uint256 _roundNumber
    ) public payable {
        require(msg.sender == owner, "[PAYMAN] Only owner can add winners");
        uint256 allocated = 0;

        for (uint256 i = 0; i < _victors.length; i++) {
            winners[_roundNumber].push(Winner(_victors[i], _gains[i]));
            balances[_victors[i]] = balances[_victors[i]] + _gains[i];
            allocated = allocated + _gains[i];
        }
```

The issue is that the winners list can be big so that the logic of the code can OOG and admins can set all the winners in one transaction. The code doesn't

support setting part of the winner's rewards for a specific round.

Recommendations

Allow setting a subset of specific round winners in one transaction.

[M-06] Plant's empire can change by front-running in purchase transactions

Severity

Impact: Medium

Likelihood: Medium

Description

In functions `createShip()`, `placeAcid()` and `chargeShield()` user defines `planetId` and the code performs the action for that planet and its empire but the code doesn't verify the empire of the plant:

```
function createShip(
    bytes32_planetId,
    uint256_overrideCount
  ) public payable _onlyNotGameOver {
    PlanetData memory planetData = Planet.get(_planetId);
    require(planetData.isPlanet, "[OverrideSystem] Planet not found");
    require(
      planetData.empireId!=EEmpire.NULL,
      "[OverrideSystem]Planetisnotowned"
    );
    uint256 cost = LibPrice.getTotalCost
      (EOverride.CreateShip, planetData.empireId, _overrideCount);
    require(_msgValue() >= cost, "[OverrideSystem] Insufficient payment");
    bytes32 playerId = addressToId(_msgSender());

    LibOverride._purchaseOverride(
      playerId,
      EOverride.CreateShip,
      planetData.empireId,
      _overrideCount,
      _msgValue
    )
```

The issue is that the empire of that plant can change after the user sends the transaction (when combats are resolved). So during battles when users want to support their empire and purchase overrides for their planets, will end up supporting the opposing empire. An attacker can front-run the user's

transaction and trigger `resolveCombat()` and change the empire of the user-defined plant to perform the attack or the issue can accrue by itself during high load and time intensive battles.

Recommendations

Get the empire in the function's input and verify that the plant belongs to that empire.

[M-07] `clearLoop()` can OOG because of the nested loop over empires and planets

Severity

Impact: Low

Likelihood: High

Description

Function `clearLoop()` loops through empires and planets and deletes their records:

```
for (uint8 i = 1; i <= empireCount; i++) {
      EEmpire empire = EEmpire(i);
      for (uint256 j = 0; j < planets.length; j++) {
        PendingMove.deleteRecord(planets[j]);
        Magnet.deleteRecord(empire, planets[j]);
      }
```

There are 6 empires and 2500 planets in the system so the total iteration of the loop would be 15000. Because in each iteration function changes the contract's storage value the gas in each iteration can be higher than 2000 and the total required gas would be higher than 30M and it can't be executed in one transaction.

Recommendations

divide the work between multiple transaction by clearing part of the data in each transaction.

[M-08] OOG error in `clearLoop()`

Severity

Impact: High

Likelihood: Low

Description

Function `ResetClearLoopSubsystem.clearLoop()` calls `PointsMap.clear(empire)` to clear all utilities associated with an empire. The issue is that this function loops through all players of the empire and clears their data and if the player's count is very big then the execution can encounter OOG.

```
function clear(EEmpire empire) internal {
    bytes32[] memory players = keys(empire);
    for (uint256 i = 0; i < players.length; i++) {
      Value_PointsMap.deleteRecord(empire, players[i]);
      Meta_PointsMap.deleteRecord(empire, players[i]);
    }
    Keys_PointsMap.deleteRecord(empire);
    Empire.setPointsIssued(empire, 0);
  }
```

Recommendations

Add restriction to the number of players or avoid looping through all of them in one transaction.

[M-09] Vulnerability to MEV sandwich attacks in point purchase and sale system

Severity

Impact: Medium

Likelihood: Medium

Description

The current design of the point system, where the price of points increases with each purchase and decreases with each sale, makes it vulnerable to MEV (Maximal Extractable Value) sandwich attacks. MEV bots can monitor the mempool and front-run legitimate buyers or sellers.

- **Front-running a buyer:** MEV bots can detect a point purchase in the mempool, purchase points ahead of the legitimate buyer, causing the price to increase. As a result, the legitimate buyer is forced to pay a higher price for their purchase and the bot can then sell at a higher price.

- **Front-running a seller:** MEV bots can detect a point sale in the mempool, and sell their points ahead of the legitimate seller at a higher price, causing the price to decrease. This would lead to the legitimate seller receiving a lower sale value for their points. The bot can then buy back at a reduced price, extracting value at the expense of the legitimate seller.

This dynamic can severely affect user experience by increasing costs for buyers and decreasing returns for sellers. It poses a significant risk, especially in high-value transactions.

Recommendations

To mitigate the risk of MEV sandwich attacks, it is recommended that slippage protection mechanisms be introduced in both purchase and sale functions. This would allow users to specify the minimum or maximum acceptable price when buying or selling points, protecting them from significant price changes caused by MEV bots.

[M-10] `placeMagnet` does not check if the planet is owned by the empire

Severity

Impact: Medium

Likelihood: Medium

Description

The `placeMagnet` function in `OverrideMagnetsSystem.sol` adds a magnet to a planet, which influences the ship's movement. To place a magnet players have to lock points to prevent them from moving enemy ships away from their home planet when they are about to attack.

However, the function does not check if the `_planetId` is owned by the `_empire`. This means that players can place a magnet on an enemy planet while obtaining and locking points in their own empire.

Players will potentially be able to affect the movement of enemy ships without locking points in the enemy empire. As the movement of ships is handled off-chain, it is uncertain if it is checked that the planet is owned by the empire for the magnet to have an effect, which is why the impact of the issue is considered "Medium".

In case the off-chain infrastructure checks that the planet is owned by the empire, players will still be able to place a magnet on an enemy planet that they expect to conquer, so that the magnet starts affecting the movement of ships as soon as they take over the planet. There might be incentives for placing the magnet before owning the planet if it is expected that the price of the override will increase in the future.

Recommendations

```
function placeMagnet(
-   EEmpire _empire,
    bytes32 _planetId,
    uint256 turnDuration
- ) public payable _onlyNotGameOver _notDefeated(_empire) {
+ ) public payable _onlyNotGameOver {
+   PlanetData memory planetData = Planet.get(_planetId);
+   EEmpire _empire = planetData.empireId;
+   require(_empire != EEmpire.NULL, "[OverrideSystem] Planet is not owned");
```

Note that the `_notDefeated` modifier is not needed because we ensure that the empire owns the planet and defeated empires do not own any planet.

8.3. Low Findings

[L-01] Incorrect game over check in `_checkTimeVictory` function

The game is considered ended when `block.number` is greater than or equal to the `P_GameConfig.getGameOverBlock()` value. This check is performed correctly in the `_onlyNotGameOver` modifier of the `EmpiresSystem` contract.

```
File: EmpiresSystem.sol

12:   modifier _onlyNotGameOver() {
13:     require(Ready.get(), "[EmpiresSystem] Game not ready");
14:     require(WinningEmpire.get
  () == EEmpire.NULL, "[EmpiresSystem] Game over");
15:     uint256 endBlock = P_GameConfig.getGameOverBlock();
16: @>  require
  (endBlock == 0 || block.number < endBlock, "[EmpiresSystem] Game over");
17:     _;
18:   }
```

However, the `_checkTimeVictory` function in the `RewardsSystem` contract does not check if the game is over correctly. This function is meant to obtain the winning empire once the game is over, however it will fail to do so if called when `block.number` is equal to the `P_GameConfig.getGameOverBlock()` value.

```
File: RewardsSystem.sol

71:   function _checkTimeVictory() internal view returns (EEmpire) {
72:     uint256 endBlock = P_GameConfig.getGameOverBlock();
73: @>  if (endBlock == 0 || block.number <= endBlock) {
74:       return EEmpire.NULL;
75:     }
```

It is recommended to apply the following change to the `_checkTimeVictory` function:

```
-     if (endBlock == 0 || block.number <= endBlock) {
+     if (endBlock == 0 || block.number < endBlock) {
```

[L-02] Wrong value assigned to the `ethSpent` field of the `PlaceAcidOverrideLog` table

The `OverrideAcidSystem.placeAcid` function registers the data of the purchase in the `PlaceAcidOverrideLog` off-chain table. The `ethSpent` field is assigned the value returned by the `_msgValue()` function, which is the amount of ether sent with the transaction. However, this value can be higher than the actual amount spent by the player, so the `cost` should be used instead.

```
File: OverrideAcidSystem.sol

37    PlaceAcidOverrideLog.set(
38      pseudorandomEntity(),
39      PlaceAcidOverrideLogData({
40        playerId: playerId,
41        turn: Turn.getValue(),
42        planetId: _planetId,
43 @>     ethSpent: _msgValue(),
44        overrideCount: 1,
45        timestamp: block.timestamp
46      })
47    );
```

[L-03] `PostManager.record()` allows skipping rounds

In the `PayoutManager` contract, the `record` function requires that the `_roundNumber` is greater than the `lastRound`. This makes it possible to skip rounds and also prevent future calls to the function if `_roundNumber` is set to `type(uint256).max`.

While the function is called by a trusted party, it is recommended to ensure that the `_roundNumber` is one greater than the `lastRound` to prevent skipping rounds by mistake. In the case there is a need to skip rounds, it be can be done explicitly by calling the function with `_victors = [address(0)]` and `_gains = [0]`.

```
require(
-           _roundNumber > lastRound,
-           "[PAYMAN] Round number must be greater than last round"
+           _roundNumber == lastRound + 1,
+           "[PAYMAN] Round number must be one greater than last round"
        );
```

[L-04] Game uses predictable randomness to select a winner

When two empires state is equal then the game uses `pseudorandom()` to determine the winner. The issue is that the result of `pseudorandom()` is predictable in each block and users can trigger winner selection logic in different blocks to change the winner. The losing empire can delay others' transactions by one or two blocks to change the winner.

[L-05] There are two rewards claiming systems that are not compatible

There are two reward distributing systems for the game winners in the current code which are not compatible with each other. One issue is that there isn't enough functionality to withdraw pot and use PayoutManager to distribute the rewards. The other issue is that between taking the snapshot and withdrawing funds from the game to use in PayoutManager users can call `withdrawEarnings()` and withdraw their funds from the game. The code should have the functionality to withdraw funds and take a snapshot of the winner's state at the same time.

[L-06] Shield Eater can be detonated on unowned planets

The `detonateShieldEater` function in the `OverrideShieldEaterSystem` contract does not check if the planet is owned by an empire before detonating the shield eater. Given that the override cost for the EEmpire.NULL empire is 0, if `detonateShieldEater` is called when the current shield eater planet is not owned by any empire, the player will not pay any marginal override cost.

```
contract OverrideShieldEaterSystem is EmpiresSystem {
  function detonateShieldEater() public payable _onlyNotGameOver {
    bytes32 playerId = addressToId(_msgSender());
    PlanetData memory planetData = Planet.get(ShieldEater.getCurrentPlanet());
+   require
+ (planetData.empireId != EEmpire.NULL, "[OverrideSystem] Planet is not owned");
```

[L-07] In delegated calls excess ether is refunded to the delegator instead of the caller

The `EmpiresSystem` contract contains the `_refundOverspend` function which refunds to `_msgSender()` the excess ether sent to the contract.

```
function _refundOverspend(uint256 _cost) internal {
    uint256 msgValue = _msgValue();
    require(msgValue >= _cost, "[EmpiresSystem] Incorrect payment");
    if (msgValue > _cost) {
      IWorld(_world()).transferBalanceToAddress(
        EMPIRES_NAMESPACE_ID,
@>      _msgSender(),
        msgValue - _cost
      );
    }
  }
```

The override systems inherit from `EmpiresSystem` and call the `_refundOverspend` function in their respective functions for purchasing overrides.

Given that the excess ether sent by the caller is refunded to `_msgSender()`, in the case of a delegated call with `World.callFrom()`, the excess ether will be refunded to the delegator instead of the caller, which may not be the intended behavior.

Option 1: Enforce that the ether sent is equal to the cost of the override.

Option 2: Add a parameter `refundAddress` to the override functions and send the excess ether to that address.

[L-08] Users cannot sell their points in some scenarios

When users want to sell their tokens by calling `sellPoints()` the code calculates the value and also updates the points price:

```
function sellPoints
    (EEmpire _empire, uint256 _points) public _onlyNotGameOver {
    uint256 pointSaleValue = LibPrice.getPointSaleValue(_empire, _points);

    // require that the pot has enough ETH to send
    require(pointSaleValue <= Balances.get(
      pointSaleValue<=Balances.get

    ), "[OverrideSystem] Insufficient funds for point sale"

    // remove points from player and empire's issued points count
    LibPoint.removePoints(_empire, playerId, _points);

    // set the new empire point price
    LibPrice.sellEmpirePointPriceDown(_empire, _points);
--snip
```

The issue is that during value calculation or price update code won't allow selling tokens if the price reaches the minimum price:

```
require(

              currentPointPrice >= config.minPointPrice + pointPriceDecrease * wholePoi
      "[LibPrice] Selling points beyond minimum price"
    );
```

The price of points increases with each buy and decreases the same amount with each sale in most situations users would able to sell all their points. But the point's price decreases overtime when admin calls `turnEmpirePointPriceDown()` so as a result point's price would reach the minimum price when there are some points left and some users won't be able to sell their points even if so contract has enough balance.

Recommendations

Instead of reverting when the price reaches the minimum limit, the code should allow selling for the minimum price and doesn't change the price.

[L-09] Attacker can perform reentrancy and cause harm for off-chain logic

In most of the purchase functions code returns the extra ETH by calling `_refundOverspend()` and then the code sets the logs to use later by the off-

chain logics.

```
--snip--
    AcidPlanetsSet.add(empire, _planetId, P_AcidConfig.getAcidDuration() - 1);
    LibOverride._purchaseOverride
      (playerId, EOverride.PlaceAcid, empire, 1, _msgValue());

    _refundOverspend(cost);
    _takeRake(cost);

    PlaceAcidOverrideLog.set(
      pseudorandomEntity(),
      PlaceAcidOverrideLogData({
        playerId: playerId,
--snip--
```

An attacker can reenter during the external call and perform other actions and as a result, the logs would be in a different order than actual actions. There's a global nonce for orders that increase with each order in `pseudorandomEntity()` which off-chain logics can determine logs order by this value. By performing this attack the nonce value and emitting order would be wrong for actions' logs.

```
function pseudorandomEntity() returns (bytes32) {
  uint256 nonce = Nonce.get();
  Nonce.set(nonce + 1);
  return bytes32(keccak256(abi.encodePacked
    (nonce, block.timestamp, block.prevrandao, block.number)));
}
```

For example attacker can buy points and sell them during reentrancy and the logs would show the attacker sold points first and then bought them.

Set logs before the external call.