Super Sushi Samurai Game

Security Review

Cantina Managed review by:

Kurt Barry, Lead Security Researcher

r0bert, Security Researcher

Sujith Somraaj, Associate Security Researcher

April 18, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Super Sushi Samurai is a social strategy focused idle fully on-chain game, played on the telegram app and powered by the Blast network

From Apr 8th to Apr 17th the Cantina team conducted a review of sss-game-contracts on commit hash 7e647a31. The team identified a total of **34** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 3
- Medium Risk: 3
- Low Risk: 16
- Gas Optimizations: 0
- Informational: 12

3 Findings

3.1 High Risk

3.1.1 `setControllerAddresses()` call can be front-run to drain the whole Diamond balance

Severity: High Risk

Context: BossHuntFacet.sol#L30, CharacterFacet.sol#L40

Description: After the contract deployment, once all the facets are added into the Diamond, a set of function calls are needed in order to configure the different facet parameters. One of these functions that should be called by the contract owner is `setControllerAddresses()`:

```
function setControllerAddresses(address createAccountController, address bossHuntController, address
↪  megaWarController) onlyOwner external {
    s.createAccountSigner = createAccountController;
    s.bossHuntAccountSigner = bossHuntController;
    s.megaWarController = megaWarController;
}
```

This function sets the `createAccountSigner` and `bossHuntAccountSigner` addresses which will be initially the zero address. The signature verification is performed using `ecrecover` directly. See `_verifyBossRaidSignature()` and `_verifyCreateAccountSignature()` functions.

However, when `ecrecover` is called, it performs a signature verification process, comparing the input signature with the calculated signature. If they match, the `ecrecover` function returns the address of the signer. **If they don't match or if the signature is invalid, the function returns the zero address.**

Consequently, as there are no checks to invalidate a zero address being returned by the `ecrecover` call, the following attack vector is possible:

1. Owner deploys the Diamond contract and adds the different facets to it.
2. `createAccountSigner` and `bossHuntAccountSigner` will be equal to the `address(0)` at this point.
3. Attacker abuses `createAccountSigner`'s zero address to create a valid account through the `registerAccount()` function by simply providing an invalid signature.
4. Attacker abuses `bossHuntAccountSigner`'s zero address to summon a boss through the `summonBoss()` function by again providing an invalid signature. This boss is summoned with a `rewardAmount` equal to the Diamond contract `SSS` token balance and a `bossPower` of 1.
5. Boss is defeated right away by the attacker who calls `claimBossReward()` to drain the contract claiming all the `SSS` tokens.

Impact: If certain steps during contract deployment are not executed atomically and in the right order, all the `SSS` token balance can be drained from the Diamond contract.

Likelihood: Medium + Impact: Very high = Severity: High.

Recommendation: It is recommended to use the ECDSA library instead of directly using `ecrecover`. This library will always revert if the `address(0)` is returned when providing an invalid signature. Moreover, make sure that no SSS tokens are sent to the Diamond contract before `createAccountSigner` and `bossHuntAccountSigner` addresses are set.

3.1.2 Users can self-approve to join any clan

Severity: High Risk

Context: MegaWarFacet.sol#L146-L166

Description: When a clan is created, the clan creator can choose if users will be required an approval to join the clan:

```
function createNewClan(uint256 landId, uint256 minMemberLevel, uint256 maxMemberLevel, bool needApproveMember)
↪  onlyNotPaused external {
    _createNewClan(landId, minMemberLevel, maxMemberLevel, needApproveMember);
}
```

This is set in the `needApproveMember` parameter. However, the `MegaWarFacet.approveMembers()` function does not check that the current approver is the actual clan creator or already a member of the clan. Consequently, any user can request to join any clan through the `joinClan()` function and then self-approve to join the clan through the `approveMembers()` function.

Impact: This can be abused to manipulate war results as a user could create multiple accounts, join the rest of the clans with these "inactive" accounts that will never level up, until reaching the max. number of members in all the other clans, forcing active users to join his clan. As the current implementation does not allow to leave a clan, the malicious user would put his own clan in a very good position to win the war and its respective rewards that season.

Likelihood: High + Impact: Medium = Severity: High.

Recommendation: Add access control to the `MegaWarFacet.approveMembers()` function by enforcing that only the clan owner or the owner of the `landId` where the clan was created can approve pending join requests.

3.1.3 Incorrect tax calculation in `teleportToland`

Severity: High Risk

Context: CharacterFacet.sol#L313

Description: The `teleportToland` function allows a player to teleport to a different land by paying the current land owner a tax. However, the tax amount is incorrectly calculated based on the land tax percentage of the destination land (`landId`) instead of the current staying land (`stayingLandId`).

```
function teleportToland(uint256 landId) onlyValidAccount onlyNotPaused external {
  // ...
  uint256 taxAmount = pointsCanUse * getLandTax(landId) / BASE_PERCENT;
  recordLandTax(stayingLandId, taxAmount);
  // ...
}
```

Recommendation: To fix the incorrect tax calculation, update the code to use the correct land tax percentage based on the current staying land (`stayingLandId`) instead of the destination land (`landId`).

```
- uint256 taxAmount = pointsCanUse * getLandTax(landId) / BASE_PERCENT;
+ uint256 taxAmount = pointsCanUse * getLandTax(stayingLandId) / BASE_PERCENT;
```

3.2 Medium Risk

3.2.1 Land owner can increase the land tax at any time forcing all the users in the land to pay the tax

Severity: Medium Risk

Context: LandFacet.sol#L53-L58

Description: The land owner can change the land tax at any given time by simply calling the `setLandTax()` function:

```
function setLandTax(uint256 landId, uint256 newTaxRate) onlyLandOwner(landId) onlyNotPaused external {
    // Tax should be <= 10%
    if(newTaxRate > 10_00) revert OutOfRange("LANDTAX");
    s.landInfo[landId].tax = newTaxRate;
    emit ChangeLandTax(landId, newTaxRate);
}
```

However, this functionality allows multiple abusive behaviours by the land owner:

1. The land owner can set a land tax of zero, wait for the land to be full of users and then call `setLandTax()` setting the land tax to the maximum allowed tax (10%).
2. Before the land owner calls `claimPoint()`, a call to `setLandTax(<landId>, 0)` is performed to avoid paying the land tax. Then after, that `claimPoint()` call, call to `setLandTax()` again setting the land tax to its previous value. This can be done atomically in a single transaction.

Impact: Land owner can avoid paying taxes in his own land. Users are forced to pay the maximum land tax even if they never accepted it.

Likelihood: Medium + Impact: Medium = Severity: Medium

Recommendation: Consider adding a cooldown period for the land owners to call `setLandTax()`. For example, only allow them to call it once per month.

3.2.2 Per-type and per-item purchase limits can be exceeded on the first purchase of an item

Severity: Medium Risk

Context: LibAppStorage.sol#L344-L365, ShopFacet.sol#L23

Description: The `addToBag()` function enforces per-item-type and per-item limits, if configured, on user purchases, but *only* if the user already holds at least one of the item being purchasing. Because `buyItemFromShopWithPermit()` allows purchasing multiple items at once, limited only by the user's token balance and the price of the item, users can exceed these limits when purchasing an item they don't already hold. For items that don't make sense to hold multiple of, for example a specific character or pet skin, users would be wasting funds if they purchased more than one due to accident or misunderstanding. Any future game mechanic that relies on these item limits would also be broken as a result.

Recommendation: Enforce per-type and per-item limits regardless of whether the user already holds the item being purchased.

3.2.3 Centralization risk: Specific `landIds` can be minted to steal the accrued rewards from the users

Severity: Medium Risk

Context: LandFacet.sol#L86

Description: In the `LandFacet` the function `withdrawStakingReward()` can only be called by the current land owner and is used to withdraw the rewards accrued by the land:

```
function withdrawStakingReward(uint256 landId) onlyLandOwner(landId) onlyNotPaused external {
    // check info
    Land storage land = s.landInfo[landId];
    uint256 lastRedeemTime = land.lastRedeemTime;
    uint256 lockTo = lastRedeemTime + s.landRedeemCountdownDurationInSecond;
    if(lastRedeemTime > 0 && block.timestamp < lockTo) {
        revert StillLocked(lockTo);
    }

    land.lastRedeemTime = block.timestamp;

    (uint256 points, uint256 reward) = LibAppStorage.stakingWithdraw(address(uint160(landId)));
    // transfer pending reward to msg.sender
    SafeERC20.safeTransfer(IERC20(s.sssToken), msg.sender, reward);
    emit LandWithdrawRewardFromPool(landId, points, reward);
}
```

This function calls `LibAppStorage.stakingWithdraw()` casting the `landId` to an address. Consequently, the following scenario would be possible:

1. Alice's address (0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) which decimal representation is 1317064453908861503093592744126318198394025057168 has staked in the contract.
2. The owner of the `Land` NFT contract mints himself the nftId 131706445390886150309359274412631819839402505716
3. The owner of the 1317064453908861503093592744126318198394025057168 `landId` calls `withdrawStakingReward(1317064453908861503093592744126318198394025057168)` stealing all the rewards from Alice.

Impact: All the user rewards can be stolen if the `Land` contract is compromised or owned by a malicious user.

Likelihood: Low + Impact: High = Severity: Medium

Recommendation: Consider enforcing in the `Land` contract that no `landId` higher than 555 can ever be minted.

3.3 Low Risk

3.3.1 Lack of mechanism to withdraw developer fee from shop revenue

Severity: Low Risk

Context: ShopFacet.sol#L60

Description: In the `_buyItemFromShop()` function, a portion of the shop transaction revenue is allocated to a "*dev*" share, stored in the `s.shopTotalTokenRevenueForDev` variable. However, no function or mechanism is provided to allow the withdrawal or distribution of these funds to the actual developers or the contract owner.

This means that the developer revenue generated from the shop transactions will be permanently locked in the contract, and the developers will not be able to access or use these funds unless a new facet is added to the diamond.

```
contract ShopFacet {
    .// ..
    s.shopTotalTokenRevenue += cost;
    s.shopTotalTokenRevenueForDev += devAmount;
    // ...
}
```

Recommendation: To address this issue, consider implementing a function that allows the withdrawal of the developer revenue from the shop transactions

3.3.2 Validate clan member levels are valid during clan creation

Severity: Low Risk

Context: MegaWarFacet.sol#L72

Description: The `_createNewClan()` function in `MegaWarFacet` allows the creation of new clans, but it does not perform any validation on the provided `minMemberLevel` and `maxMemberLevel` parameters. The function assumes that these values are valid, but there is no check to ensure that the `minMemberLevel` is less than or equal to the `maxMemberLevel`.

Additionally, the function does not enforce any limits on the minimum and maximum samurai levels. The current implementation allows samurai to create and level up to only level 50.

Recommendation: Consider adding the following checks in the `_createNewClan` function:

1. Ensure that the minMemberLevel is less than or equal to the maxMemberLevel:

```
require(minMemberLevel <= maxMemberLevel, "mg: minMemberLevel must be <= maxMemberLevel");
```

2. Ensure that the minMemberLevel and maxMemberLevel are within the expected range (e.g., between 1 and 50):

```
require(minMemberLevel >= 1 && minMemberLevel <= 50, "mg: minMemberLevel must be between 1 and 50");
require(maxMemberLevel >= 1 && maxMemberLevel <= 50, "mg: maxMemberLevel must be between 1 and 50");
```

3.3.3 Registering account with self-referral

Severity: Low Risk

Context: CharacterFacet.sol#L163, LibAppStorage.sol#L292

Description: The `_registerAccount` function in the codebase allows users to register a new account, providing an `accountOwner` address and a referrer address as parameters. However, the function does not check if the `accountOwner` and referrer addresses are the same.

This means users can register a new account and select themselves as the referrer. Users could exploit this self-referral functionality to earn referral fees on their purchases, which may not be the intended behavior.

Recommendation: Though the registration process happens off-chain, adding a defensive check in the `_registerAccount` function is recommended to prevent such issues. This can be done by adding the following conditions:

```
require(addresses[0] != addresses[1], "Referrer cannot be the account owner");
```

This will prevent users from self-referring and earning referral fees on their purchases, ensuring that the referral system is used as intended.

3.3.4 Lack of referral fee withdrawal mechanism

Severity: Low Risk

Context: ShopFacet.sol#L64, AppStorage.sol#L29

Description: The game allows users to purchase items from the shop using the ShopFacet. Whenever a purchase is made, 5% of the purchase amount is paid to the account referrer. The `totalTokenAmountFromReferral` state variable mapped to each account tracks these referral values.

However, no mechanism exists for referrers to withdraw the accumulated referral fees from the contract. This can permanently lose these funds, as the referrers cannot claim their earned referral fees.

```
contract ShopFacet {
  // ...
  function _buyItemFromShop(address accountOwner, uint256 itemType, uint256 itemId, uint256
↪  willingToPayAmount) internal {
        // ...
        if(refAccount.createdTime != 0) {
            refAccount.totalTokenAmountFromReferral += refAmount;
            _proccesReferralPetSpeedBuff(reffererAddress, refAccount);
        } else {
            burnAmount += refAmount;
        }
      // ...
    }
}
// ...
```

Recommendation: Consider implementing a mechanism that allows referrers to withdraw the accumulated referral fees from the contract.

3.3.5 `maxLandId` is configurable but the code would break if it were ever changed

Severity: Low Risk

Context: ConfigFacet.sol#L58, LandFacet.sol#L63, LandFacet#L41, LibAppStorage.sol#L129

Description: The `maxLandId` value can change, but in several places the code clearly lacks logic to deal with this possibility. The two indicated calculations in LandFacet.sol (lines 63 and 41) can cause reversion due to underflow if `maxLandId` is ever increased enough to make `s.LandTaxRewardPoolTotalPoints / s.config[1].maxLandId` less than the tax collected from one or more lands. OTOH, the check on line 129 of LibAppStorage.sol will be incorrect if `maxLandId` is ever decreased (as lands will exist with higher ids but be incorrectly classified as user accounts). This list of breakages may not be exhaustive, but demonstrates how the code is unprepared to deal with changes to `maxLandId`.

Recommendation: Consider making it impossible for `maxLandId` to change after it is set, or modify the code to be able to deal with changes to it. For example, the calculation in `LandFacet.stakeAllPoints()` could be modified to:

```
uint256 canClaimFromPoolPoints =
    pointsPerLandInThePool > s.landInfo[landId].totalPointsCollectedFromPool
        ? pointsPerLandInThePool - s.landInfo[landId].totalPointsCollectedFromPool
        : 0;
```

which would allow tax points to be staked even if no points are available to claim from the pool.

The check in `LibAppStorage.sol` could be modified to use a new variable that denotes the maximum possible maximum land if it is envisioned `maxLandId` might decrease (though this seems unlikely and problematic in other ways, e.g. for users holding land NFTs).

3.3.6 Uses of ERC20 permits are vulnerable to griefing via frontrunning

Severity: Low Risk

Context: ShopFacet.sol#L23, CharacterFacet.sol#L56, CharacterFacet.sol#L77, CharacterFacet.sol#L236

Description: The usages of the `permit` ERC20 extension in the code can be frontrun by any party that witnesses transactions prior to block inclusion. This is a griefing vector that will cause user transactions to fail if the associated `permit` has already been consumed. See this for more details. While the risk of this seems quite low (very little incentive to perform the attack, and if the centralized sequencer is trustworthy, it will not grief users or reveal their transaction inappropriately), it could be quite annoying for users, although ultimately they will be able to work around it.

Recommendation: Consider implementing an `allowance` check as a fallback if `permit` validation fails, so that transactions for which the permit was already consumed do no revert. Or, at least document the risk and justify why no action is deemed necessary.

3.3.7 Showdown winner selection randomness is extremely weak

Severity: Low Risk

Context: ShowdownFacet.sol#L89

Description: A "random" number is used to decide the winner of a showdown:

```
uint256 random = uint256(keccak256(abi.encodePacked(block.timestamp, block.coinbase, block.gaslimit,
↪  block.number, showdown.opener, showdown.attacker, showdown.openTime, showdown.attackTime)));
```

This randomness seems almost dangerously weak. An attacker will know `showdown.opener`, `showdown.attacker`, and `showdown.openTime` with certainty. Data from Blast block explorers suggests `block.gaslimit` is currently fixed at 30,000,000. `showdown.attackTime` will be the same as `block.timestamp`, and that value can be predicted to belong to a fairly small set of possibilities (blocks on Blast are only a few seconds apart, and transaction inclusion is high-probability). `block.number` is also basically known/under control of the attacker. The only remaining input is `block.coinbase`; Blast seems to use a centralized sequencer currently, so this value is likely a constant as well. Even if multiple values are possible, it's probably not a huge set.

All things considered, it's likely quite easy for attackers to write bots that calculate which showdowns they can `bluff` at any given time with a much higher probability of success than the code intends. If they don't like their odds, they simply wait a few seconds. This will greatly favor users that attack rather than start showdowns.

Recommendation: Use stronger randomness that is less under control of the attacker. Assuming Blast supports the `PREVRANDAO` opcode correctly, that is an option (as it is based on a fork of Optimism, it should have the same modified support that Optimism does).

3.3.8 No mechanism to withdraw `MegaWar` fees

Severity: Low Risk

Context: MegaWarFacet.sol#L78, AppStorage.sol#L257

Description: The game contains logic for collecting a fee when a clan is created, which is tracked by the `megawarTotalFeeCollected` storage field. The is a storage field called `megawarTotalFeeWithdrawn` that is presumably for tracking withdrawals of this fee, but there is no mechanism for actually withdrawing it or increasing the withdrawal-tracking field. Thus the fee, if collected, is lost.

Recommendation: Add a mechanism to withdraw this fee, or eliminate it.

3.3.9 `sssToken` burnt by the `ShopFacet` instead of `shopToken`

Severity: Low Risk

Context: ShopFacet.sol#L69

Description: Users pay for items from the shop using the `shopToken` ERC20. A cut of all purchase proceeds is reserved to benefit the referrer of the user making the purchase; if there is no referrer, this amount is burned instead. However, the burn call is made to the `sssToken` ERC20. In the current game design, `shopToken` and `sssToken` are actually the same, so there are no negative consequences at the moment. However, if these tokens ever differ in the future, the code will be wrong and result in erroneously burning `sssToken` instead of `shopToken`.

Recommendation: Use the `shopToken` ERC20 exclusively in the ShopFacet, or eliminate `shopToken` and use only `sssToken`.

3.3.10 Staking account counter increased and decreased inconsistently

Severity: Low Risk

Context: LibAppStorage.sol#L105, LibAppStorage.sol#L150

Description: Both normal users and lands can have staking deposits. The `stakingAccoutCount` variable is incremented every time any new address stakes, but it is only decremented when normal users (not lands) unstake. While `stakingAccoutCount` has no function within the current codebase, this inconsistency could cause issues for any UIs, bots, or other contracts that rely on this value.

Recommendation: Make the incrementing and decrementing logic for `stakingAccoutCount` consistent.

3.3.11 Direct usage of `ecrecover` allows signature malleability

Severity: Low Risk

Context: BossHuntFacet.sol#L137, CharacterFacet.sol#L180

Description: The `_verifyBossRaidSignature()` and `_verifyCreateAccountSignature()` functions call the Solidity `ecrecover` function directly to verify the given signatures. However, the `ecrecover` EVM opcode allows malleable (non-unique) signatures and thus is susceptible to replay attacks. Although a replay attack is not possible in both functions, ensuring the signatures are not malleable is considered a best practice (and so is checking `_signer != address(0)`, where `address(0)` means an invalid signature).

Impact: Unfollowed best practice.

Likelihood: Low + Impact: Low = Severity: Low

Recommendation: It is recommended to use the recover function from OpenZeppelin's ECDSA library for signature verification.

3.3.12 Lack of a two-step transfer ownership pattern

Severity: Low Risk

Context: OwnershipFacet.sol#L13-L16

Description: The ownership transfer process for the Diamond contract involves the current owner calling the `transferOwnership()` function. If the nominated EOA account is not a valid account, it is possible that the owner may accidentally transfer ownership to an uncontrolled account thereby losing access to all functions with the `onlyOwner` modifier.

All the Diamond setter functions use the `onlyOwner` modifier.

Impact: All `onlyOwner` functions may become unusable.

Likelihood: Very Low + Impact: High = Severity: Low

Recommendation: It is recommended to implement a two-step ownership transfer where the owner nominates a new owner and the nominated account explicitly accepts ownership. This ensures the nominated EOA account is a valid and active account. This can be achieved by using an implementation similar to the OpenZeppelin's Ownable2Step library.

3.3.13 Lack of incentives to keep a pet in a "*healthy*" state

Severity: Low Risk

Context: CharacterFacet.sol#L257

Description: In the `CharacterFacet` the function `feedPet()` is used to feed a pet in order to avoid the pet entering in a sick state:

```
// feed food for pet to increase live time
function feedPet(uint256 foodId) onlyValidAccount onlyNotPaused external {
    Account storage account = s.accounts[msg.sender];
    LibAppStorage.removeFromBag(msg.sender, FOOD_TYPE_ID, foodId, 1);

    uint256 petLevel = account.petLevel;
    if(petLevel < 1) revert OutOfRange("PETLEVEL");

    // Check if pet is sick, dont check sick pet
    // if (account.petDeathTime < block.timestamp) revert SickPet();

    // decrease 1% duration for each level of pet, assume max level is 50
    uint256 foodDuration = s.foodEffectDurations[foodId];
    foodDuration = foodDuration * (101 - petLevel) / 100;

    // duration is not accumulated
    account.petDeathTime = block.timestamp + foodDuration;
    emit FeedPet(msg.sender, foodId, block.timestamp + foodDuration);
}
```

If a pet is sick the following penalties are applied:

1. Shop referral speed buff will not be given.
2. It's speed is reduced by 90%.
3. Can not be leveled up.

Moreover, if the pet is sick and is given food, it will enter into a healthy state automatically and its new `petDeathTime` will be set as `account.petDeathTime = block.timestamp + foodDuration` instead of being set as `account.petDeathTime += foodDuration`. Consequently, the optimal approach for users will always be:

1. Max. out samurai level without caring about the pet level.
2. Feed the pet (if its sick) and right away leveling up the pet to max.
3. Wait for the `levelUpUnlockTime` to be reached.
4. Feeding the pet again before calling `claimLevelUpSpentToken()` and `claimPoints()`.

For this whole process only 2 food items at worst would have to be bought by the users from the shop.

Impact: Users will only buy food from the shop in the situation mentioned.

Likelihood: High + Impact: Very low = Severity: Low

Recommendation: Consider updating the `feedPet()` function as shown below:

```
- account.petDeathTime = block.timestamp + foodDuration;
+ account.petDeathTime += foodDuration;
```

This would force the users to level up their pet as soon as possible and also to buy more food from the shop.

3.3.14 Some rewards may be lost if `gameStakingPoolPercent + landPoolPercent` is not equal to `BASE_PERCENT`

Severity: Low Risk

Context: LibAppStorage.sol#L29-L44, ConfigFacet.sol#L127-L138

Description: The following state variables are used just for tracking purposes:

- `bossPoolTaxTotalReward` (increased in `reward * s.bossPoolPercent / BASE_PERCENT`).
- `megawarPoolTaxTotalReward` (increased in `reward * s.megawarPoolPercent / BASE_PERCENT`).

However, they are assigned a percentage of the total rewards in the `setTaxRewardSubPercentConfig()` function:

```
function setTaxRewardSubPercentConfig(
    uint256 gamePoolPercent,
    uint256 landPoolPercent,
    uint256 bossPoolPercent,
    uint256 megawarPoolPercent
) onlyOwner external {
    // total for game is 1.6%
    s.gameStakingPoolPercent = gamePoolPercent;
    s.bossPoolPercent = bossPoolPercent;
    s.megawarPoolPercent = megawarPoolPercent;
    s.landPoolPercent = landPoolPercent;
}
```

If `gameStakingPoolPercent + landPoolPercent` is not equal to `BASE_PERCENT`(100_00) due to some fraction being added to `bossPoolPercent` or `megawarPoolPercent` a portion of the rewards collected from the community taxes will remain in the contract and not be distributed:

```
function syncTaxOfGameFromToken() internal {
    AppStorage storage s = appStorage();
    if(ISSS(s.sssToken).communityTaxTokenAmountAvailable()> 0) {
        try ISSS(s.sssToken).claimCommunityTax() returns (uint256 reward) {
            // Split the reward for: staking pool, land, boss, megawar
            uint256 rewardForStaking = reward * s.gameStakingPoolPercent / BASE_PERCENT;
            s.stakingNewReward += rewardForStaking;
            s.landPoolTaxTotalReward += reward * s.landPoolPercent / BASE_PERCENT;
            s.bossPoolTaxTotalReward += reward * s.bossPoolPercent / BASE_PERCENT; // <----------------
            s.megawarPoolTaxTotalReward += reward * s.megawarPoolPercent / BASE_PERCENT; // <---------
            s.afkGamePoolTaxTotalReward += rewardForStaking;
        } catch {
            // do nothing
        }
    }
}
```

Impact: Lower accrued rewards for users. A portion of the rewards collected from the community taxes will remain in the contract and not be distributed.

Likelihood: Low + Impact: Medium = Severity: Low

Recommendation: Consider enforcing at smart contract level that `gameStakingPoolPercent + landPoolPercent` is equal to `BASE_PERCENT`.

3.3.15 `_proccesReferralPetSpeedBuff()` does not consider the amount of items being bought

Severity: Low Risk

Context: ShopFacet.sol#L84-L107

Description: In the `ShopFacet` the function `_proccesReferralPetSpeedBuff()` is used to calculate the speed buff that the referrer will receive after an item was purchased from the shop. However, the the amount of items being bought are not taken into account during the calculation meaning that buying 3 items will result in the same speed buff than buying one (as long as this is done in a single `buyItemFromShop()` call). However, if, for example, the 3 items are bought in 3 separate `buyItemFromShop()` calls, the referral will receive the speed buff with every call resulting in an speed buff 3 times higher that if he had just called `buyItemFromShop()` once buying the 3 items.

Impact: Referral speed buff is not applied correctly when more than one item is bought.

Likelihood: High + Impact: Very low = Severity: Low

Recommendation: It is recommended to take into account also the amount of items being bought in the speed buff calculation in the `_proccesReferralPetSpeedBuff()` function.

3.3.16 First user calling `claimPoint` will receive all the accrued rewards since contract deployment until the second call to `claimPoint`

Severity: Low Risk

Context: LibAppStorage.sol#L46-L71

Description: The function `stakingUpdatePool()` is used to update the `stakingAccRewardPerShare` state variable:

```
function stakingUpdatePool() internal {
    AppStorage storage s = appStorage();

    // Get reward from token tax pool
    syncTaxOfGameFromToken();

    if(s.stakingTotalDepositSupply == 0) {
        return;
    }

    (uint256 reward, bool needUpdated) = getRewardWithPromotion();
    if(needUpdated) {
        s.promotionUpdatedBlock = block.number;
    }
    // overflow check
    // max reward is 10^52 = (total deposit supply * 10^18) * REWARDS_PRECISION < 2^256
    // MAX_POINT_PER_SECOND = 400000 * 10^18
    // MAX_POINT_PER_YEAR = 400000 * 10^18 * 365 * 24 * 60 * 60 = 0.126144 * 10^32
    // if there is 10M (10^7) of players, MAX_POINT_PER_YEAR = 0.126144 * 10^39
    // then stakingTotalDepositSupply is still less than 2^256
    uint256 rewardPerShare = reward * REWARDS_PRECISION / s.stakingTotalDepositSupply;
    if(rewardPerShare > 0) {
        s.stakingAccRewardPerShare += rewardPerShare;
        s.stakingNewReward = 0;
    }
}
```


However, `stakingAccRewardPerShare` will only be increased once `stakingTotalDepositSupply` is different than zero. And this will only occur after the first call to the `claimPoint()` function. Consequently, the first user that calls the `claimPoint()` function will receive all the accrued rewards since contract deployment.

Impact: "Unfair" distribution of the initial accrued rewards.

Likelihood: High + Impact: Very low = Severity: Low.

Recommendation: Consider updating the `stakingUpdatePool()` function logic to prevent this issue.

3.4 Informational

3.4.1 Declare `petTypeId` and `samuraiTypeId` as constants

Severity: Informational

Context: CharacterFacet.sol#L108, CharacterFacet.sol#L355, CharacterFacet.sol#L363, LibAppStorage.sol#L335

Description: To add an element to a bag, there is a unique item type and ID. However, the item types for Samurai and Pets are identified as `1` and `2`, respectively. Instead of declaring these values as constants, they are declared/hardcoded as magic numbers in multiple instances across the entire codebase.

For example, In the `changeset` and `changeSamurai` functions, the petTypeId and samuraiTypeId values are declared as 2 and 1, respectively. These values are used to access the player's bags and check if the player owns the specified pet or samurai.

Since these values are not expected to change throughout the contract's lifetime, it would be more efficient and logical to declare them as constant variables.

Recommendation: Consider declaring `petTypeId` and `samuraiTypeId` as constant variables at the contract level and then using these constants in the `changePet`, `changeSamurai`, and `createAccount` functions. This makes the code more readable, maintainable, and less prone to errors.

```
+ uint256 constant PET_TYPE_ID = 2;
+ uint256 constant SAMURAI_TYPE_ID = 1;
```

3.4.2 Add zero checks to prevent creation of empty locks

Severity: Informational

Context: File.sol#L123

Description: The current Lock contract implementation needs checks to prevent the creation of empty locks, where the contract is deployed without any initial funds. This can lead to a situation where the contract is deployed, but no funds can be withdrawn, as the contract balance is 0.

Recommendation: To prevent the creation of empty locks, consider adding a check in the constructor to ensure that the contract is deployed with a non-zero balance. This can be done by adding a require statement that checks the initial contract balance:

```
  constructor(uint256 _unlockTime) payable {
      require(block.timestamp < _unlockTime, "Unlock time should be in the future");
+     require(msg.value > 0, "Contract must be deployed with non-zero balance");
      unlockTime = _unlockTime;
      owner = payable(msg.sender);
  }
```

3.4.3 Use `call` Instead of `transfer` for native token transfers

Severity: Informational

Context: Lock.sol#L32

Description: The current implementation uses the `transfer` function to send a native token from the `Lock` contract to an external address. The transfer function is limited to 2300 gas, which may not be enough for all contract interactions.

This can lead to potential issues, such as the transfer failing if the receiving contract has a fallback function that requires more than 2300 gas to execute.

Recommendation: Instead of using the `transfer` function, consider using the `call` method to send native tokens from the contract to an external address. The call function does not have the 2300 gas limit and allows for more complex contract interactions.

```
  function withdraw() public {
    require(block.timestamp >= unlockTime, "You can't withdraw yet");
    require(msg.sender == owner, "You aren't the owner");

    emit Withdrawal(address(this).balance, block.timestamp);
-   owner.transfer(address(this).balance);
+   payable(owner).call{value: address(this).balance}("");
  }
```

3.4.4 Remove unused/debugging helper file imports

Severity: Informational

Context: CharacterFacet.sol#L7, BossHuntFacet.sol#L5-L8, ConfigFacet.sol#L7-L9, LandFacet.sol#L8-L11, MainGameFacet.sol#L6-L7, MegaWarFacet.sol#L7, MegaWarFacet.sol#L10, ShopFacet.sol#L8, ShopFacet.sol#L5, ShowdownFacet.sol#L5, ShowdownFacet.sol#L7-8, ShowdownFacet.sol#L10-12, AppStorage.sol#L3, LibAppStorage.sol#L2, LibAppStorage.sol#L5, Lock.sol#L5, Lock.sol#L25

Description: Multiple files across the entire repository contain an import statement not used anywhere in the contract (or) and used for debugging purposes during development.

Importing unused libraries can increase the contract's deployment and execution gas costs and make the codebase less readable and maintainable.

Recommendation: Consider removing the unused and debugging file imports from multiple contracts across the repository to optimize the gas costs and improve the code quality.

3.4.5 Return if `claimId` (or) `seasonId` is zero in `_recordClanLevel()`

Severity: Informational

Context: CharacterFacet.sol#L123

Description: The `_recordClanLevel` function does not check for cases where the decoded `clanId` and `seasonId` from the `currentClanInfos` function are zero. In that case, the function will still attempt to access the `s.megaWars` and `s.clans` mappings, which does not impact the game/user.

Hence, returning if the `clanId` and `seasonId` are zero is more logical.

Recommendation: To mitigate this potential issue, consider adding a check at the beginning of the `_recordClanLevel` function to ensure that the decoded `clanId` and `seasonId` are not zero before proceeding with the rest of the function logic.

```
if (clanId == 0 || seasonId == 0) {
    return;
}
```

3.4.6 Add missing license identifiers

Severity: Informational

Context: BossHuntFacet.sol, CharacterFacet.sol, ConfigFacet.sol, LandFacet.sol, MainGameFacet.sol, MegaWarFacet.sol, RewardPoolFacet.sol, ShopFacet.sol, ShowdownFacet.sol, BlastFacet.sol, AppStorage.sol, LibAppStorage.sol, Modifiers.sol, UniswapV2Helper.sol

Description: All Solidity files should include a license identifier at the top to communicate the licensing terms under which the code is distributed. This helps ensure legal compliance and transparency for users and contributors.

The lack of a license identifier could lead to ambiguity and potential legal issues, as the default copyright laws may apply, which may not align with the project's intended licensing terms.

The license identifiers are added to each facet in the diamond reference implementation by Nick Mudgen.

Recommendation: Consider adding a license identifier, such as `SPDX-License-Identifier: MIT` or another appropriate license, at the top of each Solidity file in the project.

3.4.7 Replace deprecated `block.difficulty` in `weakRandom()`

Severity: Informational

Context: LibAppStorage.sol#L158

Description: The current implementation of the weakRandom function uses `block.difficulty` as one of the inputs to the pseudo-random number generation. However, `block.difficulty` has been deprecated and is no longer recommended for use in favor of `block.prevrandao`.

Recommendation: Consider using 'block.prevrandao`instead of`block.difficulty' if the blast sequencer offers support.

Blast, being based on a fork of Optimism, likely uses the same PREVRANDAO implementation as Optimism, although this should be confirmed before usage as the `prevrandao` values are set by the sequencer:

```
  function weakRandom(bytes memory seed) internal view returns (uint256) {
-   return uint256(keccak256(abi.encodePacked(block.timestamp, block.difficulty, msg.sender, seed)));
+   return uint256(keccak256(abi.encodePacked(block.timestamp, block.prevrandao, msg.sender, seed)));
  }
```

3.4.8 Underflow in `findUpperBound` can cause an unintended revert

Severity: Informational

Context: LibAppStorage.sol#L476

Description: The `findUpperBound` function opens with a series of checks:

```
function findUpperBound(uint256[] storage array, uint256 element) internal view returns (uint256) {
    uint256 len1 = array.length - 1;
    if (array.length == 0 || array[0] < element) {
        return 0;
    }
    if(array[len1] > element) {
        return array.length;
    }
    // ...
```

The computation of `len1` will cause a revert due to underflow if `array.length == 0`, contradicting the intention of the next check to return 0 in this case.

Recommendation: Move the computation of `len1` after the explicit `array.length == 0` check:

```
- uint256 len1 = array.length - 1;
  if (array.length == 0 || array[0] < element) {
      return 0;
  }
+ uint256 len1 = array.length - 1;
```

3.4.9 `decodeUserClanInfo` uses an incorrect shift value

Severity: Informational

Context: LibAppStorage.sol#L531

Description: `decodeUserClanInfo()` is implemented as follows:


```
function decodeUserClanInfo(uint256 info) internal pure returns (uint256 clanId, uint256 seasonId) {
  clanId = info >> 64;
  seasonId = (info << 224) >> 224;
}
```

Strictly speaking, the shift used for the `seasonId` should only be 192 (since $64 + 192 = 256$). While it is almost certainly impossible for `seasonId` to reach 2^{32} in practice, if that ever happened this would return the wrong value.

Recommendation: Use 192 instead of 224 for the `seasonId` shifts to be strictly correct.

3.4.10 Minor code quality issues

Severity: Informational

Context: AppStorage.sol#L89, AppStorage.sol#L206, CharacterFacet.sol#L372, CharacterFacet.sol#L310

Description: The codebase contains minor typos and naming inconsistencies that should be addressed for better code quality and maintainability.

AppStorage.sol

```
struct LevelSpeeAndCost { // typo: LevelSpeedCost (d is missing)
  // ...
}

struct AppStorage {
  // ...
  uint256 stakingAccoutCount; // stakingAccountCount(n is missing)
}
```

CharacterFacet.sol

```
contract CharacterFacet {
  // ...
  (,uint256 totalPointAvaible) = LibAppStorage.recordAvailablePoint(msg.sender); // totalPointAvailable
↪  (Avaible -> Available)
  // ...
  function caclculateMaxPlayerPerLand(uint256 totalPlayer) { // calculateMaxPlayerPerLand
  // ...
  }
}
```

Recommendation: Consider fixing the typos and naming inconsistencies identified in the code to improve code quality and maintainability.

3.4.11 Land's occupation will be determined exclusively by their tax

Severity: Informational

Context: CharacterFacet.sol#L288

Description: With the current game implementation, the attractiveness of a Land is only determined by its tax and of course limited by the current players in that Land as there is a maximum of 50 players per land. Consequently, the land with the lowest tax will probably be the land with most players as that is the land that will maximize users rewards.

This is because when the points are claimed in the `claimPoint()` function a tax that is sent to the land owner is applied. This `taxAmount` is calculated as `pointsCanUse * getLandTax(landId) / BASE_PERCENT`. Users will always try to be in the land with the lowest tax to stake as many points as possible:

```
// Claim point and move it to the staking pool
function claimPoint() onlyValidAccount onlyNotPaused external {
    Account storage account = s.accounts[msg.sender];
    if (account.petLevel == 0) revert InvalidAcount();

    if(account.startUnlockSpentTokenTime != 0) revert StillLocked(account.levelUpUnlockTime);

    // Calculate pending coin: samurai point
    // Calculate Available point: pet point
    (, uint256 totalPending) = LibAppStorage.recordPendingPoint(msg.sender);
    (, uint256 totalAvailablePoint) = LibAppStorage.recordAvailablePoint(msg.sender);

    // Get min of totalPending and totalAvailablePoint
    uint256 pointsCanUse = Math.min(totalPending, totalAvailablePoint);

    account.pointPending -= pointsCanUse;
    // account.pointAvailable -= pointsCanUse;
    account.pointAvailable = 0; // reset available point to 0

    // Calculate land-reward based on the land tax setting
    // 10% of land-reward goes to the land pool
    // 90% of land-reward goes to the land owner
    uint256 landId = account.stayingAtLandId;
    uint256 taxAmount = pointsCanUse * getLandTax(landId) / BASE_PERCENT;
    recordLandTax(landId, taxAmount);

    // Move total point to reward pool
    LibAppStorage.stakingDeposit(msg.sender, pointsCanUse-taxAmount);

    emit ClaimPoint(msg.sender, pointsCanUse - taxAmount, taxAmount);
}
```

Impact: Lands occupation will be determined exclusively by their tax. Land with the lowest tax will always be full.

Likelihood: High + Impact: Very low = Severity: Informational

Recommendation: Consider adding some extra game rules that affect the attractiveness of a land apart from its land tax.

3.4.12 The pre-image of `DIAMOND_STORAGE_POSITION`'s storage slot is known

Severity: Informational

Context: LibDiamond.sol#L31

Description: The preimage of the hashed storage slot `DIAMOND_STORAGE_POSITION` is known.

Impact: Unfollowed best practice.

Likelihood: High + Impact: None = Severity: Informational

Recommendation: It might be best to subtract 1 so that the preimage would not be easily attainable. As an example, this is the technique that OpenZeppelin uses.