CANTINA

Sheepcoin
Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Rustyrabbit, Security Researcher

March 7, 2025

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

Sheepcoin is is a novel GameCoin project, Its basic premise is that sheep left idle will be targets for the hungry wolves.

The security review focused on examining the core contracts responsible for the AMM functionality, token management, fee collection, and governance mechanisms. The audit team assessed the codebase for potential vulnerabilities that could lead to fund loss, economic exploits, or other security issues.

From Feb 26th to Mar 1st the Cantina team conducted a review of SheepCoin on commit hash 535c1d52. The team identified a total of **18** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	1	1	0
Low Risk	5	2	3
Gas Optimizations	0	0	0
Informational	10	5	5
Total	**18**	**10**	**8**

3 Findings

3.1 High Risk

3.1.1 First Depositor Inflation Attack in SheepDog Contract

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The `SheepDog` contract is susceptible to a "first depositor inflation attack" where an attacker can exploit the share allocation mechanism to gain disproportionate control over other users deposits. By calling `protect` with a minimal deposit (e.g., 1 wei) when `totalShares` is 0, the attacker mints 1 share and becomes the sole shareholder. They then directly transfer a large quantity of SHEEP tokens to the contract via an ERC20 transfer, inflating the `totalsheep` balance without altering `totalShares`. Subsequent legitimate deposits, especially those smaller than the inflated totalsheep balance, receive 0 shares due to precision loss in the `uint256` calculation `_amount * totalShares / totalsheep`, which rounds down to zero as `totalShares` is too small relative to `totalsheep`. After a 2-day withdrawal delay, the attacker can claim the entire pool, including all deposited SHEEP, leaving other users with no returns.

Recommendation: Consider implementing one of the following solutions:

1. Mint initial shares to a dead address: When deploying the `SheepDog` contract, mint a predetermined, fixed amount of shares (e.g., 10000 shares) and assign them to a dead address like address(0). These shares should be permanently locked, meaning they cannot be redeemed or transferred. This ensures that the `totalShares` (or equivalent total supply variable) starts at a non-zero value.
2. Set a minimum initial deposit threshold: Require depositors to contribute a sufficiently large amount of assets (e.g., a minimum `totalsheep` value). This prevents the initial deposit from being too small, which could exacerbate rounding errors or allow manipulation.

Ceazor Snack Sandwich: Fixed in 40b898d.

Cantina: Fix OK.

3.1.2 Incorrect Swap Amount in buySheep Function

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `SheepDog` contract's `buySheep` function, the swap operation uses an incorrect token amount, leading to a guaranteed revert every time this function is called. The function calculates `balGasToken` as the total `wGasToken` balance of the contract and determines `buyAmount` as `balGasToken` minus a 5% team fee. However, when invoking `IRouter(router).swapExactTokensForTokensSimple`, it specifies `balGasToken` as the amount to swap instead of `buyAmount`. This mismatch occurs because the router is only approved to spend `buyAmount`, yet the swap attempts to use the full `balGasToken` balance, which exceeds the approved amount. As a result, the transaction will revert due to insufficient allowance, preventing the contract from purchasing SHEEP tokens as intended and disrupting the yield mechanism for depositors.

Recommendation: Correct the swap operation in the `buySheep` function by replacing `balGasToken` with `buyAmount` in the `IRouter(router).swapExactTokensForTokensSimple` call to align the swapped amount with the approved amount.

Ceazor Snack Sandwich: Fixed in 36539b3.

Cantina: Fix OK.

3.2 Medium Risk

3.2.1 Users can pool SHEEP to reduce `sheepDog` fee

Severity: Medium Risk

Context: sheepDog.sol#L98-L101

Description: Rent for protecting SHEEP by depositing them into the `sheepDog` contract is charged as 10 wGAS per user per day regardless of how many SHEEP is deposited.

A group of users can pool their sheep in a contract, let `sheepDog` protect all of them and only pay 10wGAS per day for all users combined.

It's also far cheaper for a user holding a large amount of sheep compared to one that holds a smaller amount.

Recommendation: Although this could be regarded as part of the gamification, consider charging a fee per SHEEP per day rather than per user.

Ceazor Snack Sandwich: Fixed in fda3c0f.

Cantina: Fix OK.

3.3 Low Risk

3.3.1 Insufficient Pre-Mint Limit Check in SHEEP Contract

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `SHEEP` contract's `_mintForFee` function, the check:

```
require(preMinted < maxPreMint, "No more sheep in the market");
```

fails to adequately enforce the maximum pre-mint supply cap, allowing the total supply of SHEEP tokens to exceed the intended limit. This condition only verifies that the current value of `preMinted` is less than `maxPreMint` before adding the requested `_amount`. However, it does not account for the size of `_amount` itself, meaning a single large mint request could push `preMinted` beyond `maxPreMint` without triggering a revert. For instance, if `maxPreMint` is 2,000,000e18 and `preMinted` is 1,000,000e18, a call to mint 2,000,000e18 would pass the check (since 1,000,000e18 < 2,000,000e18) and result in `preMinted` equaling 3,000,000e18, overshooting the cap by 1,000,000e18.

Recommendation: Update the pre-mint limit enforcement by modifying the condition to:

```
require(preMinted + _amount <= maxPreMint, "No more sheep in the market");
```

in the `_mintForFee` function. This adjustment ensures that the sum of the current `preMinted` value and the new mint amount does not exceed `maxPreMint`.

Ceazor Snack Sandwich: Fixed in 52e1060.

Cantina: Fix OK.

3.3.2 Burn Limit in WolfNFT Contract Allows Four Consecutive Burns from SheepMarket

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `WolfNFT` contract, the condition:

```
require(eatenFromMarket[_wolfID] <= 3 || _victim != sheepMarket, "you eat too much from the market");
```

in the `eatSheep` function inaccurately enforces the intended limit of three consecutive burns from the `sheepMarket` per wolf, allowing up to four burns instead. The use of <= permits the wolf to execute a burn when `eatenFromMarket[_wolfID]` equals 3, incrementing it to 4, because the check passes as long as the value is less than or equal to 3 at the time of evaluation. This means a wolf can burn SHEEP from the sheepMarket four times consecutively (starting at 0, then 1, 2, and 3), exceeding the apparent design goal of restricting burns to a maximum of three.

Recommendation: Update the condition to:

```
require(eatenFromMarket[_wolfID] < 3 || _victim != sheepMarket, "you eat too much from the market");
```

to strictly limit each wolf to three consecutive burns from the `sheepMarket`.

Ceazor Snack Sandwich: Fixed in 31cf281.

Cantina: Fix OK.

3.3.3 Static Rent Fee Misalignment with SHEEP Token Value in SheepDog Contract

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `SheepDog` contract, the rent fee for protecting `SHEEP` tokens is defined as a static constant, `TEN`, set to `10 * 1e18` wGAS tokens per day, as calculated in the `getCurrentRent` function. This fixed fee does not adjust based on the market value of `SHEEP` tokens, creating a potential economic imbalance. If the price of `SHEEP` rises significantly relative to `wGAS`, the 10 `wGAS` daily fee becomes negligible, making protection overly affordable and reducing the intended cost-benefit trade-off for users. Conversely, if `SHEEP`'s value drops, the fee becomes disproportionately expensive, deterring users from utilizing the `SheepDog` for protection.

Recommendation: Consider updating `TEN` from being a `constant` to a mutable state variable that can be updated by the contract's owner, allowing the rent fee to be adjusted in response to changes in `SHEEP`'s market value. On the other hand, introduce an `onlyOwner` function, such as `setRentFee(uint256 newFee)`, that enables the owner to periodically update the rent fee based on `SHEEP`'s price relative to `wGAS`.

Ceazor Snack Sandwich: Acknowledged.

3.3.4 Daily Rent Calculation Loses Partial-Day Precision

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `SheepDog` contract, the rent calculation in the `getCurrentRent` function, defined as:

```
uint256 _calcRent = (block.timestamp - rentStart[_user]) / 86400 * TEN;
```

suffers from precision loss due to integer division, resulting in undercharging users for partial days. The division by 86400 (seconds in a day) occurs before multiplication by `TEN` (10 `wGAS` tokens), causing any fractional day to be truncated. For example, if a user withdraws after 2 days and 12 hours (216,000 seconds), the calculation (216,000 / 86,400) yields 2 days, and the rent remains 20 `wGAS` until the elapsed time reaches 3 full days (259,200 seconds). This allows users to withdraw anytime between exactly 2 days and just under 3 days while still paying only for 2 days, providing up to nearly a full day of free protection. This deviates from an intent where rent might be expected to reflect partial days.

Recommendation: Update the rent calculation in the `getCurrentRent` function to account for partial days by multiplying before dividing, ensuring fractional time periods contribute to the rent fee:

```
function getCurrentRent(address _user) public view returns (uint256) {
    uint256 secondsElapsed = block.timestamp - rentStart[_user];
    // Multiply first, then divide -> partial-day rent is included
    // TEN is 10e18, i.e. 10 wGAS with 18 decimals, so rent can be fractional
    return (secondsElapsed * TEN) / 86400;
}
```

Ceazor Snack Sandwich: Acknowledged.

3.3.5 Wolf Burn Mechanism Can Be Abused by Splitting SHEEP Across Multiple Wallets

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `SheepCoin` ecosystem, `SHEEP` token holders can initially mitigate the risk of wolf burns by splitting their holdings into multiple wallets, each containing an amount less than `ONE` (1 * 10ˆ18 wei, or 1 `SHEEP` token), exploiting a limitation in the `WolfNFT` burn mechanics.

The `eatSheep` function in the `WolfNFT` contract burns an amount of `SHEEP` equal to `hunger[_wolfID]`, which starts at `ONE` for a newly minted wolf and increases by `ONE` with each burn. Early in the game, when wolves have a hunger level of `ONE`, they can only burn `SHEEP` from addresses holding at least 1 full `SHEEP` token; if an address holds less than this amount (e.g., 0.999 `SHEEP`), the burn fails due to insufficient balance, as the ERC20 transfer or burn operation would revert when the requested amount exceeds the available balance. By distributing `SHEEP` across multiple wallets in sub-`ONE` amounts, holders can initially avoid burns entirely. This strategy becomes more effective over time as wolves burn `SHEEP` repeatedly, increasing their hunger levels (e.g., to 2, 3, or more `SHEEP` per burn), requiring larger splits to stay below the threshold. This long term game exploit undermines the intended threat posed by wolves, reducing the immediate incentive to use the `SheepDog` contract for protection and potentially affecting negatively the ecosystem's risk-reward balance during its end phase.

Recommendation: A possible solution would be to allow wolves to eat multiple times from different addresses, decrementing their `hunger` until it reaches 0, at which point it is increased by `ONE` again. This would enable wolves to consume fractional amounts across multiple wallets in a single feeding cycle, reducing the effectiveness of the splitting strategy while preserving the burn mechanic's threat.

Ceazor Snack Sandwich: Acknowledged. This possibility is noted and the added complexity of the UX was considered more troublesome than worth it. New Wolfs can be minted to address holding less than the lowest hunger wolf. The value of 0.99 SHEEP is so low, that the gas cost of spreading so much might even make it not worth it to do so, not considering the time to disperse and manage.

3.4 Informational

3.4.1 Lack of Cleanup Mechanism for Starved WolfNFTs

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `WolfNFT` contract, the condition:

```
require(block.timestamp < starved[_wolfID], 'your wolf starved');
```

in the `eatSheep` function prevents a wolf from burning `SHEEP` tokens if its starvation timer (`starved[_wolfID]`) has expired, effectively rendering it unable to perform its core function. However, there is no mechanism to burn, destroy, or otherwise clean up these starved WolfNFTs, leaving them minted and owned indefinitely despite their useless state. This results in a growing number of inactive NFTs cluttering the contract, as a starved wolf cannot `eatSheep` and provides no further utility to its owner.

Recommendation: Implement a cleanup mechanism to remove starved WolfNFTs from circulation by adding a function, such as `burnStarvedWolf(uint256 _wolfID)`, that allows owners or the contract to burn a wolf NFT once its `starved[_wolfID]` timestamp has passed. This function should verify the starvation condition before calling `_burn(_wolfID)` to destroy the NFT and update relevant state variables. To incentivize participation, the game could introduce a reward mechanism within this function, such as minting a small amount of `SHEEP` tokens (e.g., a fixed reward or a percentage based on the wolf's past burns) to the owner upon burning a starved wolf, encouraging users to clean up their inactive NFTs.

Ceazor Snack Sandwich: Acknowledged. A pile NFT here contract can be deployed later to reward users that had wolves and took them to certain levels of hunger. This finding was noted.

3.4.2 Repeated Forced Burns from the Liquidity Pool Deter and Harm LPs

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `SheepCoin` ecosystem, the `WolfNFT` contract enables a mechanism where `SHEEP` tokens are burned directly from the `sheepMarket` liquidity pool. This process is designed to reduce the circulating supply of `SHEEP`, potentially driving up its price. However, this mechanism poses a significant problem for liquidity providers in the `sheepMarket`, as the frequent burns could outweigh the benefits of providing liquidity, making participation in the pool economically unattractive.

When `SHEEP` tokens are burned directly from the liquidity pool, they are removed without a corresponding withdrawal of the paired token (e.g., `wGAS`). This disrupts the balance of the pool, artificially increasing the price of `SHEEP` within it. Here's how this creates issues for LP'ers:

- Impermanent loss amplification: Each burn shifts the pool's balance, and the subsequent arbitrage trades further adjust the ratio of `SHEEP` to the paired token. This leads to persistent impermanent loss for LP'ers, as their share of the pool's total value diminishes with every cycle of burns and trades.

- Cumulative effect of frequent burns: With many burns occurring over time, the repeated disruption and value extraction can accumulate, significantly eroding the pool's value for LP'ers. This cumulative impact could easily outweigh the benefits of providing liquidity, such as the trading fees earned, rendering the role of LP'er unsustainable.

Liquidity providers are essential to the `sheepMarket`, as they ensure there's enough liquidity for trading activity. However, if the direct burning of `SHEEP` tokens consistently undermines their economic incentives, LP'ers may withdraw their assets from the pool. Without adjustments to protect or compensate LP'ers, this design flaw threatens the long-term viability of the `sheepMarket` liquidity provision model.

Recommendation: Remove or strictly limit the ability to burn directly from the pair. Alternatively, introduce a compensation mechanism where a portion of each burn's value (e.g., a fee in `wGAS` or a minting of additional `SHEEP` tokens) is redistributed to LP'ers as a reward for their exposure to burn-related risks.

Ceazor Snack Sandwich: Acknowledged. Participants are going to be highly encouraged to NOT provided liquidity and to allow POL to be the sole LPs.

3.4.3 Sheep Burns Can Be Sandwiched

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `WolfNFT` contract's burn mechanism, which removes `SHEEP` tokens from the `sheepMarket` liquidity pool when triggered by a wolf owner, is susceptible to front-running and sandwich attacks due to its public and predictable nature. An attacker can monitor the mempool for pending burn transactions and exploit the anticipated price increase of `SHEEP` tokens. Specifically, the attacker submits a buy transaction with a higher gas price to purchase `SHEEP` before the burn executes, then sells the tokens at a profit after the burn reduces the supply and drives up the price. In a sandwich attack, the attacker surrounds the burn with both a buy and a sell transaction, amplifying their profits.

This attack could be executed by the owner of the `WolfNFT` that executes the burn atomically to amplify its profit.

Recommendation: Merely informative, there is no trivial fix to this issue. However, the feasibility of this attack is considered low because the amount of `SHEEP` burned by a single wolf transaction is likely to be a small fraction of the total liquidity in the `sheepMarket` pool, limiting the economic incentive and impact of front-running or sandwiching.

Ceazor Snack Sandwich: Acknowledged. This issue was noted and due to the unlikelihood of the burns being of high value, changes were not deemed necessary.

3.4.4 BuySheep swapExactTokensForTokensSimple Call Can Be Sandwiched

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `SheepDog` contract, the call to:.

```
IRouter(router).swapExactTokensForTokensSimple(balGasToken, 1e18, wGasToken, sheep, false, address(this),
↪  block.timestamp + 10);
```

within the `buySheep` function is susceptible to sandwich attacks due to its predictable nature and public visibility. An attacker can monitor the mempool for this pending `swap` transaction and manipulate the market by placing a buy order before it to increase the price of `SHEEP`, followed by a sell order after the swap executes, profiting from the price difference at the contract's expense. Additionally, the function uses a fixed `amountOutMin` parameter of `1e18` (1 `SHEEP` token), which does not dynamically adjust to the trade size. This static minimum output would not prevent this attack vector.

This directly harms the `SheepDog` yield, which relies on the `buySheep` function to convert `wGasToken` into `SHEEP` tokens for distribution as rewards to depositors. When an attacker sandwiches the swap, the contract receives fewer `SHEEP` tokens than anticipated for the same amount of `wGasToken`. As a result, the yield available for distribution decreases, reducing the rewards for depositors.

Recommendation: This issue is noted for informational purposes, as the feasibility of this sandwich attack exploit is considered quite low under typical conditions. Given that the `buySheep` function is public and expected to be called frequently, the swapped amounts (`balGasToken`) are likely to be small relative to the liquidity pool size in `sheepMarket`. Consequently, the potential profit from front-running or sandwiching these swaps would be minimal, making the attack economically unviable in most scenarios. To validate this assessment, developers should monitor the frequency of `buySheep` calls and the typical swap amounts in relation to pool liquidity, documenting this limitation to inform users of the potential risk while emphasizing its low practicality. No immediate fix is required, but awareness of this vulnerability ensures transparency within the ecosystem.

Ceazor Snack Sandwich: Fixed in 7fff587.

Cantina: Partially fixed by adding an incentive for users to call this function often. However, the sandwich attack, even if its very unlikely to ever occur, is still possible if the amount swapped is high enough.

3.4.5 Long-Term Depletion of WolfNFT Utility and Its Impact on SheepCoin Ecosystem Dynamics

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `SheepCoin` ecosystem, `WolfNFTs` serve as a mechanism to burn `SHEEP` tokens, reducing their circulating supply and driving game dynamics. However, several design features suggest a potential endgame scenario where wolves become obsolete, diminishing key incentives within the system.

Firstly, the cost of minting a `WolfNFT` increases with each purchase, as the `mating` variable increments by `ONE` (e.g., 1 SHEEP for the first wolf, 2 for the second, etc.), eventually making it economically unfeasible for players to acquire new wolves. Secondly, each time a wolf burns `SHEEP` via the `eatSheep` function, its hunger level increases by `ONE`, meaning it requires progressively more `SHEEP` for subsequent burns. This escalating requirement, combined with a starvation mechanic where wolves must eat at least once every 7 days or become permanently unable to burn `SHEEP`, suggests that over time, all wolves will either starve or become impractical to maintain due to the rising `SHEEP` cost per burn.

In this endgame, with no active wolves remaining, the burning pressure on `SHEEP` ceases, removing the primary incentive for using the `SheepDog` contract (protection from burns) and reducing the strategic value of providing liquidity in the `sheepMarket`.

Recommendation: This observation is provided for informational purposes to highlight a potential long-term evolution of the `SheepCoin` ecosystem, rather than an immediate flaw.

Ceazor Snack Sandwich: Acknowledged. This issue is noted and due to the short term nature of this project, deemed not a big issue. There is also plans to reward NFT holders pro-rata based on hunger level achieved.

3.4.6 Missing NatSpec Comments

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: All the contracts in the code-base are missing or have incomplete code documentation, which affects the understandability, auditability and usability of the code. Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables, parameters, etc. This special form is named the Ethereum Natural Language Specification Format NatSpec.

Recommendation: Consider adding in full NatSpec comments for all functions to have complete code documentation for future use.

Ceazor Snack Sandwich: Fixed in 2d42074.

Cantina: Partially solved in the given commit by adding new comments in the code explaining the different flows in the contracts. However, NatSpec comments were not added explicitly as it was suggested.

3.4.7 Lack of a Double-Step Transfer Ownership Pattern

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The standard OpenZeppelin's Ownable contract allows transferring the ownership of the contract in a single step:

```
/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public virtual onlyOwner {
    if (newOwner == address(0)) {
        revert OwnableInvalidOwner(address(0));
    }
    _transferOwnership(newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the `onlyOwner` modifier.

Recommendation: It is recommended to implement a two-step transfer process where the owner nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account. A good code example could be OpenZeppelin's Ownable2Step contract:

```
/**
 * @dev Starts the ownership transfer of the contract to a new account. Replaces the pending transfer if there
 ↪  is one.
 * Can only be called by the current owner.
 *
 * Setting `newOwner` to the zero address is allowed; this can be used to cancel an initiated ownership
 ↪  transfer.
 */
function transferOwnership(address newOwner) public virtual override onlyOwner {
    _pendingOwner = newOwner;
    emit OwnershipTransferStarted(owner(), newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`) and deletes any pending owner.
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual override {
    delete _pendingOwner;
    super._transferOwnership(newOwner);
}

/**
 * @dev The new owner accepts the ownership transfer.
 */
function acceptOwnership() public virtual {
    address sender = _msgSender();
    if (pendingOwner() != sender) {
        revert OwnableUnauthorizedAccount(sender);
    }
    _transferOwnership(sender);
}
```

Ceazor Snack Sandwich: Fixed in 02b3cbc.

Cantina: Fix OK.

3.4.8 Theoretical Reentrancy Risk in getWolf Function

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `WolfNFT` contract, the `getWolf` function contains a theoretical reentrancy vulnerability due to its use of the `_safeMint` function when minting a new wolf NFT. The `_safeMint` function, inherited from the `ERC721`, includes an external call to the recipient's `onERC721Received` hook if the recipient is a contract. This external call occurs before the function completes its state updates, such as incrementing `wolfID` and setting starved, hungry and hunger values, potentially allowing a malicious contract recipient to reenter `getWolf` or call other functions. However, this reentrancy is not exploitable in the current implementation.

Subsequent reentrant calls would mint the same `wolfID` reverting. Additionally, trying to reenter the `eatSheep` function's would also revert due to the following check:

```
require(block.timestamp < starved[_wolfID], 'your wolf starved');
```

Despite this, the theoretical risk remains a concern from a security best-practices perspective.

Recommendation: To eliminate this theoretical reentrancy risk and adhere to safer coding practices, consider replacing `_safeMint` with `_mint` in the `getWolf` function.

Ceazor Snack Sandwich: Acknowledged. This issue was noted and since its not exploitable, left as is. But in future code updates this information will be considered and taken into account.

3.4.9 `dogSleep()` waiting period intent is ambiguous

Severity: Informational

Context: sheepDog.sol#L69

Description: Whenever a users deposits their SHEEP into the `sheepDog` it is locked in the contract for a minimum of 2 days. If the user wants to withdraw their SHEEP they need to call `dogSleep()` and wait 2 days before they can actually withdraw them with `getSheep()`.

It's not clear what exactly the intended design is of the waiting period. Is it just so users cannot deposit and withdraw within a short period or should the user signal their intent to withdraw to other users and possibly even be forced to do so.

A user can call `dogSleep()` immediately after depositing them with `protect()`. After the initial 2 day waiting period the SHEEP will still be protected but the user will at any time be able withdraw their SHEEP without waiting time. The rent still needs to be payed over the full time that they are protected, but as such the `dogSleep()` function doesn't really provide any utility (compared to just not allowing them to withdraw based on the deposit time).

The only restriction it does provide is that users can no longer deposit extra SHEEP from the same address once `dogSleep()`is called. They can however still do the same from another address.

Recommendation: Depending on intended design consider enforcing the waiting period simply based on the deposit time so users do not need to call `dogSleep()` or only allow (or enforce) the user to withdraw their sheep within an allowed time window after `dogSleep()` was called (e.g.between 2 and 5 days) to ensure they have to signal their intent to withdraw at least 2 days before actually doing so.

Ceazor Snack Sandwich: Fixed in 47288a0.

Cantina: Fix OK.

3.4.10 Code improvement suggestions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Following are some suggested code quality improvements.

Set `hunger` of new wolf directly to `ONE` as `hunger[wolfID]`of a new wolf will always be 0:

- WolfNFT.sol#L61.

Use constants where possible:

- `ONE_WEEK`in SheepV3.sol#L397.

- `mintPrice`, `teamCut` and `maxPreMint`in SheepV3.sol#L400-L402.

Use immutables where possible:

- ERC20 `name` and `symbol` in SheepV3.sol#L58-L59.

Remove unused storage variables:

- `sheepToCLaim` in sheepDog.sol#L2.

Remove unused events:

- `sheepBorn`, `sheepSlaughtered`, `lassieReleased`, `sheepPastured` and `newSheppard` in SheepV3.sol#L413-L415.
- `newSheppard` in SheepV3.sol#L417.

Remove `_beforeTokenTransfer` and `_afterTokenTransfer` as they are not implemented.

- SheepV3.sol#L239.
- SheepV3.sol#L252.
- SheepV3.sol#L267.
- SheepV3.sol#L276.
- SheepV3.sol#L293.
- SheepV3.sol#L305.
- SheepV3.sol#L361.
- SheepV3.sol#L377.

Ceazor Snack Sandwich: Fixed in c46d082.

Cantina: Fix OK.