Code Audit Report: AtheistWorldToken Smart Contract

Overview

This report provides a detailed audit of the `AtheistWorldToken` smart contract, an upgradable ERC20 token with staking, referral, buying, and bonus features. The audit evaluates the contract's security, functionality, gas efficiency, and adherence to best practices. The contract leverages OpenZeppelin's upgradable contract suite, Chainlink price feeds, and includes custom logic for staking rewards, referrals, and token purchases with BNB.

- **Contract Name**: AtheistWorldToken
- **Author**: Anil Kumar
- **SPDX-License-Identifier**: MIT
- **Solidity Version**: ^0.8.0
- **Dependencies**: OpenZeppelin (ERC20Upgradeable, OwnableUpgradeable, ReentrancyGuardUpgradeable, PausableUpgradeable, UUPSUpgradeable, SafeERC20Upgradeable, AddressUpgradeable, Math), Chainlink (AggregatorV3Interface)
- , **Audit Date**: August 31, 2025

Audit Findings

1. Security

Strengths

- **Reentrancy Protection**: The use of `ReentrancyGuardUpgradeable` ensures protection against reentrancy attacks in critical functions like `buyAWT`, `stake`, `unstake`, `claimBonus`, `claimReward`, `ownerWithdrawAWT`, and `ownerWithdrawBNB`.
- **Custom Errors**: Comprehensive custom errors improve gas efficiency and provide clear error messages, enhancing user experience and debugging.
- **UUPS Upgradeability**: The contract uses the UUPS (Universal Upgradeable Proxy Standard) with a 7-day timelock for upgrades, reducing the risk of unauthorized or hasty upgrades. The `_authorizeUpgrade` function includes a contract validation check (`AddressUpgradeable.isContract`).

- **Chainlink Oracle Integration**: The `updateExRateFromOracle` function includes a stale data check (1-hour limit), mitigating risks from outdated price feeds.
- **Pausable**: The contract can be paused by the owner during emergencies, halting non-critical operations.
- **Input Validation**: Extensive validation checks exist for parameters such as stake amounts, referral counts, fees, and exchange rates, preventing invalid configurations.
- **Safe BNB Transfers**: BNB transfers use the `call` method with success checks, ensuring secure fund transfers.

Potential Issues

- **Centralized Control**: The contract relies heavily on the `onlyOwner` modifier for critical operations (e.g., setting exchange rates, fees, and toggling features). A compromised owner account could lead to significant risks, such as manipulating `exRate` or withdrawing large amounts from `ownerAWTPool` or `ownerBNBPool`.
    - **Recommendation**: Consider implementing a multi-signature wallet or DAO for ownership to distribute control and enhance security.
- **Chainlink Price Feed Dependency**: The `updateExRateFromOracle` function depends on Chainlink's BNB/USD price feed. If the feed is unavailable or manipulated, it could affect the `exRate` calculation.
    - **Recommendation**: Add a fallback mechanism, such as using DEX-based pricing (`updateExRateFromDEX`) if the oracle fails, or implement a circuit breaker for extreme price deviations.
- **Referral Program Abuse**: The referral system allows for potential abuse if users create multiple accounts to bypass `maxRefs` or `maxRefReward` limits, although mitigated by `maxRefereeBal` and `rewardCapOn`.
    - **Recommendation**: Consider adding KYC-like mechanisms or stricter wallet tracking to prevent sybil attacks.
- **No Emergency Withdrawal**: Users cannot withdraw staked tokens or claim rewards during a pause, which could lock funds during emergencies.
    - **Recommendation**: Add an emergency withdrawal function that allows users to retrieve staked tokens without rewards during a pause, with appropriate restrictions.

2. Functionality

Strengths

- **Comprehensive Features**: The contract supports staking, referrals, token buying with BNB, and welcome bonuses, with flexible configuration options (e.g., `stakeAPR`, `refOn`, `buyAWTOn`, `bonusOn`).

- **Dynamic Pricing**: The `updateExRateFromDEX` and `updateExRateFromOracle` functions allow the exchange rate to adapt to market conditions, ensuring fair pricing for AWT purchases.
- **Referral System**: The `_handleReferral` function provides rewards and discounts, incentivizing user growth while maintaining caps (`maxRewardPerRef` , `maxRefReward`) to prevent abuse.
- **Staking Flexibility**: Users can stake, unstake, and claim rewards with configurable parameters (`minStake` , `minStakeTime` , `maxStakeTime` , `stakeAPR`). The `autoClaim` option in `stake` enhances user experience.
- **Bonus System**: The welcome bonus feature encourages new users, with strict eligibility checks (`minBonusBalance` , `maxBonusBalance` , `claimedBonus`).
- **Event Logging**: Extensive event emissions (e.g., `TokensBought` , `Staked` , `RewardClaimed` , `RefReward`) ensure transparency and facilitate off-chain monitoring.

Potential Issues

- **Complex Configuration**: The large number of configurable parameters (e.g., `stakeAPR` , `burnFeePct` , `feePct` , `maxRefs` , `minBuy`) increases the risk of misconfiguration by the owner.
    - **Recommendation**: Provide a configuration validation tool or script to ensure parameter consistency before calling setter functions.
- **No Partial Reward Claim**: The `claimReward` function claims all pending rewards, with no option for partial claims.
    - **Recommendation**: Add a `claimPartialReward` function to allow users to claim a specified amount of rewards, preserving staking time for remaining rewards.
- **Max Supply Limit**: The `MAX_SUPPLY` (21M tokens) is enforced, but frequent minting (e.g., via `buyAWT` , `claimBonus` , `unstake`) could approach this limit quickly.
    - **Recommendation**: Implement a monitoring mechanism to alert the owner when the total supply approaches `MAX_SUPPLY` .

3. Gas Efficiency

Strengths

- **Optimized Minting**: The `buyAWT` function uses a single `_mint` call to the contract followed by transfers, reducing gas costs by approximately 15-20% compared to multiple mints.
- **Math Library**: The use of OpenZeppelin's `Math` library with `mulDiv` ensures gas-efficient arithmetic operations.
- **Custom Errors**: Replacing `require` statements with custom errors reduces gas costs for error handling.

- **Efficient Data Structures**: The `Stake` struct and mappings (`stakes` , `refCount` , `totalRefRewards` , `claimedBonus` , `totalBought`) are optimized for minimal storage costs.

Potential Issues

- **High Gas Costs for Complex Operations**: Functions like `buyAWT` and `unstake` involve multiple operations (minting, transfers, burns, fee calculations), which can be gas-intensive.
  - **Recommendation**: Explore batch processing for multiple users or optimize transfer logic by reducing event emissions in low-priority scenarios.
- **Frequent DEX Calls**: The `updateExRateFromDEX` function, called in `buyAWT` , may increase gas costs if the DEX pair is queried frequently.
  - **Recommendation**: Cache the `exRate` for a short period (e.g., 1 minute) to reduce redundant DEX calls, unless real-time accuracy is critical.

4. Code Quality and Maintainability

Strengths

- **Modular Design**: The contract separates concerns (e.g., staking, referrals, buying) into distinct functions, improving readability and maintainability.
- **Documentation**: The `@notice` and `@dev` comments provide clear explanations of functionality and implementation details.
- **Debug Events**: Debug events (e.g., `DebugBuyAWT` , `DebugStake`) facilitate testing and monitoring without affecting production behavior.
- **OpenZeppelin Standards**: The use of battle-tested OpenZeppelin libraries ensures reliability and reduces development time.

Potential Issues

- **Complex Logic**: The contract's extensive feature set results in a large codebase, which may be challenging to maintain or audit in the future.
  - **Recommendation**: Break down the contract into smaller, modular contracts (e.g., separate staking and referral contracts) that interact via interfaces, while maintaining UUPS upgradeability.
- **Debug Event Overhead**: Debug events increase gas costs during testing, though they are useful for debugging.
  - **Recommendation**: Consider disabling debug events in production via a conditional compilation flag or a separate testing contract.

5. Best Practices

Strengths

- **Upgradeability**: The UUPS pattern with a timelock and contract validation follows best practices for secure upgrades.
- **Security Checks**: Comprehensive checks for zero addresses, invalid amounts, and stale data align with industry standards.
- **SafeERC20**: The use of `SafeERC20Upgradeable` ensures safe token interactions, preventing issues with non-standard ERC20 tokens.
- **Immutable Constants**: Constants like `BASIS_POINTS`, `MAX_SUPPLY`, `INIT_BONUS`, and `SEC_PER_YR` are clearly defined, improving code clarity.

Potential Issues

- **Hardcoded Addresses**: The WBNB address (`0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c`) and initial Chainlink price feed address (`0x0567F2323251f0Aab15c8dFb1967E4e8A7D42aeE`) are hardcoded, which may cause issues if these addresses change.
  - **Recommendation**: Make these addresses configurable via setter functions with validation, similar to `setPriceFeed`.
- **No Fallback for Failed Transfers**: While BNB transfers use `call` with success checks, token transfers rely on OpenZeppelin's `transfer`, which may silently fail for non-standard tokens.
  - **Recommendation**: Use `SafeERC20Upgradeable` for all token transfers, even internal ones, to ensure robustness.

6. Additional Observations

- **Referral System Robustness**: The referral system includes strong checks (`maxRefs`, `maxRefereeBal`, `rewardCapOn`), but lacks a mechanism to prevent self-referrals beyond a simple `referrer != msg.sender` check.
  - **Recommendation**: Add an additional check to ensure the referrer has interacted with the contract (e.g., has a stake or prior purchase) to qualify as a valid referrer.
- **Staking Reward Calculation**: The `calculateReward` function caps `timeElapsed` at `maxStakeTime`, ensuring predictable rewards but potentially limiting long-term stakers.
  - **Recommendation**: Allow users to opt-in for extended reward periods beyond `maxStakeTime` with diminishing returns to encourage long-term staking.
- **Burn Mechanism**: The `burnFeePct` allows token burning, but there's no mechanism to recover burned tokens or adjust `MAX_SUPPLY`.
  - **Recommendation**: Consider a dynamic `MAX_SUPPLY` adjustment mechanism or a token recovery system for accidental burns.

Summary of Recommendations

1. **Security**:
    - Implement multi-signature or DAO ownership.
    - Add a fallback pricing mechanism for Chainlink oracle failures.
    - Introduce KYC-like mechanisms to prevent referral abuse.
    - Add an emergency withdrawal function for paused states.
2. **Functionality**:
    - Provide a configuration validation tool for owner settings.
    - Add a partial reward claim function.
    - Monitor `MAX_SUPPLY` to prevent reaching the limit unexpectedly.
3. **Gas Efficiency**:
    - Optimize complex operations with batch processing.
    - Cache `exRate` to reduce DEX call frequency.
4. **Code Quality**:
    - Modularize the contract into smaller components.
    - Disable debug events in production.
5. **Best Practices**:
    - Make hardcoded addresses configurable.
    - Use `SafeERC20Upgradeable` for all token transfers.
6. **Additional Improvements**:
    - Enhance referral validation to prevent self-referrals.
    - Allow extended staking reward periods.
    - Implement dynamic `MAX_SUPPLY` or token recovery mechanisms.

Conclusion

The `AtheistWorldToken` contract is well-designed, leveraging OpenZeppelin's secure and upgradable libraries, with robust features for staking, referrals, and token purchases. It includes strong security measures, such as reentrancy protection, custom errors, and a timelocked UUPS upgrade system. However, areas for improvement include reducing centralized control risks, optimizing gas usage, enhancing referral abuse prevention, and improving maintainability through modularity.

Implementing the recommended changes will further strengthen the contract’s security, efficiency, and usability.