Smart Contract Security Audit Report

For

GCE

Date Issued: September.30, 2025

Version: v1.0

Confidentiality Level: Public

Contents

1 Abstract

This report was prepared for GCE smart contract to identify issues and vulnerabilities in its smart contract source code. A thorough examination of GCE smart contracts was conducted through timely communication with GCE, static analysis using multiple audit tools and manual auditing of their smart contract source code.

The audit process paid particular attention to the following considerations.

- A thorough review of the smart contract logic flow
- Assessment of the code base to ensure compliance with current best practice and industry standards
- Ensured the contract logic met the client's specifications and intent
- Internal vulnerability scanning tools tested for common risks and writing errors
- Testing smart contracts for common attack vectors
- Test smart contracts for known vulnerability risks
- Conduct a thorough line-by-line manual review of the entire code base

As a result of the security assessment, issues ranging from critical to informational were identified. We recommend that these issues are addressed to ensure a high level of security standards and industry practice. The recommendations we made could have better served the project from a security perspective.

- Enhance general coding practices to improve the structure of the source code.
- Provide more comments for each function to improve readability.
- Provide more transparency of privileged activities once the agreement is in place.

2 Overview

2.1 Project Summary

Project Summary	Project Information
Name	GCE
Start date	September.23, 2025
End date	September.30, 2025
Contract type	DEX
Language	Solidity

2.2 Report HASH

Name	HASH
Contract.zip	39155106cd4574d6871324293bb369f0cd478636e94d4d9a5da60b1981f20ff6

3 Project contract details

3.1 Contract Overview

Token.sol

The Token contract is a burnable ERC-20 token contract (Golden Cicada Effect, GCE) with transaction tax and LP handling. Upon deployment, the contract mints the total amount to the designated owner and creates a trading pair with USDT. The trading logic is AMM-aware: 3% of purchases from the trading pair are transferred to nodeRobot. When selling to the trading pair, 25% is directly allocated from LPs to the donatePool, 5% is burned from LPs, 1% is collected from the seller to replenish LPs, and 5% is transferred to nodeRobot to trigger subsequent distributions. The contract then synchronizes to the reserve (sync). The contract supports a buy-only switch (openBuy), whitelist/blacklist exemptions (isGuardedOf), and one-time configuration of nodeRobot, burnOperator, and donatePool. There is also a strong permission function burnTokenFromPair that allows burnOperator to directly transfer and sync tokens from LP. The overall function is used to provide a source of funds for the NodeRobot/donation pool in the ecosystem and impose taxes and controls on secondary market transactions.

Achievement.sol

The Achievement contract manages performance and levels. It records individual and team performance for each address (including cumulative subchain performance), calculating small team performance by excluding the largest leg for level assessment. It provides two types of superior filters

(pointAccounts and breekAccounts) to determine the list of superiors eligible for profit sharing on the direct referral chain. It implements the diffAwards rule (searching for higher-level superiors along the ancestral path, with a total proportion $\leqslant$ 5%). It automatically determines and executes levelUpgrades based on preset thresholds (individual performance + small team performance), switching accounts to the corresponding level allocation pool via IHome.changeLevelPool. It supports performance increase by authorized roles (increaseDelegate) and direct level upgrades by node operations (upLevelByNodeFarm). All sensitive operations and access to external relationships (family tree IFamily, computing power IHome) are completed through PermissionControl and interfaces, forming a closed loop of "family relationship → performance accumulation → level assessment → level pool allocation."

Family.sol

The Family contract maintains family/invitation relationships: It constructs a single-parent chained relationship tree with rootAddress as the root, recording each address's parent (parentOf), depth (depthOf), and direct referral list (_childrenMapping). It provides read-only queries: getForefathers for tracing back up the ancestor chain at a given depth, and querying the direct referral list (childrenOf) and count (childrenCount). Relationship establishment begins when any new address binds its parent to its parent (provided the parent is already in the tree and the child address has not been previously bound) via makeRelation(parent). Internally, it increments the count, writes the parent pointer and depth, and adds the child address to the parent's direct referral list. This system is used to provide reliable chained referral/organizational relationship data for other services (performance statistics, ratings, etc.).

FomoPool.sol

The FomoPool contract is a lottery-based dividend pool that distributes USDT evenly on a daily basis. The contract is permission-controlled: accounts with the FOMO_POOL_ADD_USER role can add designated users to the current cycle based on an on-chain pseudo-random number (with a 50% probability), with a maximum of 120 participants per cycle. The onlyManager can call startDistribute to set the daily starting time. At most once per day, anyone can call distribute: half of the contract's allocable USDT (current balance minus the reserved reserve) is allocated to the perUserRewardOf function for that period, distributed evenly among all participants. The cycle number is then incremented, and this total is added to the reserve. Users can use earm to estimate their payout for up to 50 settled cycles, starting from the cursor. claim traverses up to 50 participation records starting from the user's cursor, aggregates the rewards for all settled cycles, and transfers them all at once, simultaneously advancing the cursor and decrementing the reserve accordingly. The overall closed loop of "being selected to join the cycle → half of the balance in the pool is evenly distributed every day → users receive it in batches according to the cycle" is realized.

Home.sol

The Home contract is the project's core "financial management/dividend distribution" contract: users stake USDT, and the contract converts 75% of the stake into project tokens through routing and deposits them into the donatePool. 3% is sent to nodeRobot (for node allocation), and the remaining USDT is directly injected into trading pairs and synced. The contract also records users' "computing power" (half of the principal) and

"redeemable certificates" (half of the principal) and accumulates historical deposits. Users can harvest daily returns based on computing power (calculated using a 20-day yield formula that fluctuates between 1% and 2%) after the daily settlement interval. These returns are distributed in tokens from the donatePool, and "attainment bonuses/cultivation bonuses" are distributed based on family relationships (Achievement/Family). Upon meeting certain conditions, users can unstake and redeem "redeemable certificates" with USDT through interaction. The contract also maintains "level pools" from 1 to 8. Using distributeLevelPool, token rewards estimated based on total deposits are weighted and distributed to a per capita share at each level. Users can claim them using takeLevelReward. When an achievement upgrades, the level pool status is migrated using changeLevelPool. Large participation (⩾200 USDT) will result in the user being considered for the current FomoPool lottery. This completes a closed loop of "deposit → computing power accounting → daily income and multi-dimensional rewards → redemption." Multiple contracts (Router, NodeRobot, Achievement, Interaction, and FomoPool) collaborate to handle liquidity processing, node allocation, and tiered rewards.

Interaction.sol

The Interaction contract is a USDT redemption/redeem assistance contract: only external business contracts/addresses with the INTERACTION_CLAIM_USDT role can call claim(user, amount). The contract executes swapTokensForExactTokens on the router along the Token→USDT path, swapping the project tokens held by this contract (previously granted unlimited authorization to the router in setToken) for an exact amount of USDT directly to the specified user. To prevent transaction failures, a maximum of 1.5x the estimated investment is allowed (which results in

significant slippage). After the transaction is completed, the Token contract calls burnTokenFromPair(amount, address(this)) to "pull" an equal amount of tokens from the trading pair address (and sync), replenishing the token inventory of this contract and affecting the pool reserve. The overall purpose of this contract is to convert token positions within the system into actual USDT payments to users, enabling USDT redemption and issuance in modules like Home/LpPool.

LpPool.sol

The LpPool contract is a "periodic LP staking/redemption" contract: after the user selects a 7/15/30-day term (OrderType), they call stake(amount, orderType). The contract deposits the user's USDT directly into the trading pair, then calculates the required token quantity from the user using the getAmountsIn route, transfers the required token quantity from the user, and sends it to the trading pair before calling sync(). This effectively injects bilateral liquidity into the token/USDT pool on the user's behalf (recorded as the order principal in the amount of USDT 2, order.amount). Each address can hold a maximum of 10 orders, and the account's cumulative investment in the pool (USDT2) must not exceed the total historical investment recorded by the Home contract (Home.allSupply). Upon expiration, if the user calls unstake, the contract calculates the profit (using 1e12 precision) based on the formula "principal + fixed daily interest rate × number of days" and redeems it in USDT via Interaction.claim. The order is then deleted. The contract also provides user order list query, and sets external contract addresses such as routing, tokens, Home/Interaction, and interest rate parameters for each term during initialization.

Multicall3.sol

The Multicall3 contract is a general-purpose "batch call" contract: it allows for the aggregation of multiple function calls to any target contract into a single transaction. It supports aggregate / blockAndAggregate , which must all succeed; tryAggregate / tryBlockAndAggregate / aggregate3 , which can optionally ignore failures and return the success or failure status and return value of each call; and aggregate3Value , which can attach a specified ether amount to each subcall (and verify the sum of msg.value). It also provides a set of read-only helper methods for querying the block number, block hash, timestamp, coinbase, difficulty, gaslimit, basefee, chain ID, and the native token balance of any address.

NodeFarm.sol

The NodeFarm contract is a "node equity and dividend center" contract: the project first configures the price and quantity caps for three types of nodes (King, Supreme, and Genesis). Users purchase nodes through makeNode (either in full USDT or with a "60% USDT + 40% project token" path price conversion). A one-time referral share is recorded in shareLogs according to the rules for the direct ancestors of the family tree. Each node type has a USDT dividend pool (poolOf[nodeType], accounted for by accPerUserShare). When NodeRobot or another address calls distribute to a pool, USDT is distributed evenly to each user based on the number of users currently holding USDT in the pool. Node holders can claim their accumulated USDT rewards through harvest if "Home.userPower * 2 >= node price" is met. The contract is also integrated with Achievement: purchasing a node accumulates the performance of direct referrals. Users can call activate based on this accumulated performance, triggering an external level increase

using Achievement.upLevelByNodeFarm. Administrators can airdrop Genesis nodes in batches (makeGenesisNode) or urgently repair Genesis configurations (repairGenesisNode). They can also set key dependencies such as home/achievement/nodeRobot. Overall, this contract is responsible for node creation and capacity control, linking to family relationships, recording and withdrawing USDT rewards, and interfacing with external Home/NodeRobot/Achievement.

Achievement.sol

The NodeRobot contract is the project's "fund router and distributor": after initialization, it records dependencies such as USDT, routing, NodeFarm, FomoPool, treasury (exchequer), and operating address, and pre-authorizes USDT for NodeFarm; the administrator sets the project token through setToken. Whenever the token contract sells, it transfers the fee token to this contract and calls distributeFromSell (which can only be called by token). The contract will directly convert the received token into USDT according to the route (no minimum transaction volume is set), and then divide the obtained USDT into five parts: 3 parts are injected into NodeFarm's Genesis/King/Supreme node pools, and the other 2 parts are transferred to FomoPool and the treasury respectively; then the internal _distributeFromBuy is called to convert all the remaining tokens held by it into USDT and distribute them again in the ratio of 1/3 (Genesis) + 1/6 (King) + 1/6 (Supreme) + 1/3 (operationReceiver); In addition, accounts with the NODE_ROBOT_DISTRIBUTE_FROM_BUY role can actively trigger distributeFromBuy to perform the same "buy-side" distribution. Overall, it is responsible for converting tokens from transaction fees or operational injections into USDT, and automatically distributing them to various revenue pools and operating accounts according to fixed weights.

4 Audit results

4.1 Key messages

ID	Title	Severity	Status
01	After reaching the top and clearing, the accumulated taken is inconsistent	Medium	Fixed
02	The order of reducing the total amount of _clearUserPower is wrong, causing totalSupply to never decrease	High	Fixed
03	FomoPool qualification can be cheated (more than 200 accounts can only grab 120 places)	Low	Ignore
04	Transfer USDT directly to Pair and sync() to push up the price	Low	Ignore
05	No slippage protection, easy to be squeezed by MEV/ causing tokenAmount to be raised	Low	Ignore
06	Privileged roles can modify multiple variables in the contract	Low	Fixed
07	lastIndex variable is not used	Informational	Fixed
08	No event log for key methods	Informational	Ignore

4.2 Audit details

4.2.1 After reaching the top and clearing, the accumulated taken is inconsistent.

ID	Severity	Location	Status
1	Medium	Home.sol	Fixed

Description

When the upper limit of taken ⩾ power*3 is reached, _clearUserPower() is called to clear the value to zero (including taken = 0). Subsequently, userOf[msg.sender].taked += tokenValue is executed. This causes a positive taken to appear after the value is cleared, which is inconsistent with the logic.

Code location:

```
function takeBreedReward() public {
    address[] memory pathSell = new address[](2);
    pathSell[0] = address(token);
    pathSell[1] = address(usdt);
    uint256 tokenAmount = userOf[msg.sender].breedToken;

    if (tokenAmount > 0) {
        uint256 tokenValue = router.getAmountsOut(tokenAmount, pathSell)[1];
        if (userOf[msg.sender].taked + tokenValue >= userOf[msg.sender].power * 3) {
            tokenValue = userOf[msg.sender].power * 3 - userOf[msg.sender].taked;
            address[] memory pathBuy = new address[](2);
            pathBuy[0] = address(usdt);
            pathBuy[1] = address(token);
            tokenAmount = router.getAmountsOut(tokenValue, pathBuy)[1];
            require(tokenAmount > 0, "token amount is 0");
            _clearUserPower(msg.sender);
        }
        userOf[msg.sender].taked += uint128(tokenValue);
        token.safeTransferFrom(donatePool, msg.sender, tokenAmount);
        userOf[msg.sender].breedToken = 0;
        emit Claim(msg.sender, tokenValue, tokenAmount, AwardType.Breed);
    }
}
```

Recommendation

It is recommended to modify the current logic and stop adding takend after clearing it to keep the logic reasonable and safe..

Status

Fixed.

4.2.2 The order of reducing the total amount of _clearUserPower is wrong, causing totalSupply to never decrease

ID	Severity	Location	Status
2	High	Home.sol	Fixed

Description

The current logic is to first set user.power to 0, then execute totalSupply -= userOf[user].power * 2. Since power is already 0, totalSupply will not decrease.

This way, totalSupply only increases, and the distributedLevelPool uses router.getAmountsOut(totalSupply/100, [usdt, token]) to estimate totalReward, which will be inflated over time, causing accPerUserShare to be falsely inflated. This will result in a large number of subsequent claim failures and an overdraft of the donation pool.

Code location:

```
function _clearUserPower(address user) internal {
    userOf[user].power = 0;
    userOf[user].pointToken = 0;
    userOf[user].breedToken = 0;
    totalSupply -= userOf[user].power * 2;
    userOf[user].taked = 0;
    userOf[user].accSupply = 0;
}
```

```
function distributeLevelPool() external {
    require(block.timestamp - lastDistributeLevelPoolAt >= 1 days,"not enough time");
    address[] memory path = new address[](2);
    path[0] = address(usdt);
    path[1] = address(token);
    uint256 totalReward = router.getAmountsOut(totalSupply / 100, path)[1];

    if (totalReward > 1e18) {
        lastDistributeLevelPoolAt = uint256(block.timestamp);
        for (uint256 i = 1; i <= LEVEL_POOL_COUNT; i++) {
            if (levelPoolOf[i].totalUser > 0) {
                levelPoolOf[i].accPerUserShare +=
                    (totalReward * levelPoolOf[i].rate) /
                    1e12 /
                    levelPoolOf[i].totalUser;
            }
        }
    }
}
```

Recommendation

It is recommended to first execute totalSupply -= userOf[user].power * 2, and then set user.power to 0.

Status

Fixed.

4.2.3 FomoPool qualification can be cheated (more than 200 accounts can only grab 120 places)

ID	Severity	Location	Status
3	Low	Home.sol	Ignore

Description

The same account can only be included once in the same cycle, but a large number of accounts can be used to repeatedly deposit >200e18 to trigger addUser, and cooperate with FomoPool (the randomness of FomoPool.addUser can be repeatedly brushed in by roles, and there is a miner/validator bias.) Internally unsafe randomization "grabs" all 120 places in the current period.

Code location:

```
if (amount > 200e18) {

    IFomoPool(fomoPool).addUser(msg.sender);
}
function addUser(address _user) external onlyRole(FOMO_POOL_ADD_USER) {
    uint256 randomSeed = uint256(
        keccak256(abi.encodePacked(blockhash(block.number - 1),block.timestamp,msg.sender,block.difficulty)));

    if ((randomSeed & 1) == 0 && cycleUserCountOf[currentCycle] < 120 && !isInCycle(_user, currentCycle)) {
        userCyclesOf[_user].push(currentCycle);
        cycleUserCountOf[currentCycle]++;
        emit AddUser(_user, currentCycle);
    }
}
```

Recommendation

It is recommended to increase randomness to avoid the same user getting all the results.

Status

Ignore.Customer response is logical.

4.2.4 Transfer USDT directly to Pair and sync() to push up the price

ID	Severity	Location	Status
4	Low	Home.sol	Ignore

Description

Transferring all USDT in the contract directly into Pair and sync()ing it will instantly drive up the token price. Arbitrageurs immediately sell tokens to exchange for this batch of USDT, which is equivalent to subsidizing new user deposits to the market, making it extremely vulnerable to sandwich attacks.

```
usdt.safeTransfer(pair, usdt.balanceOf(address(this)));
IPair(pair).sync();
```

Recommendation

It is recommended to limit the funds to a small amount for price balancing to avoid sandwich attacks caused by large amounts of funds.

Status

Ignore.Customer response is logical.

4.2.5 No slippage protection, easy to be squeezed by MEV/ causing tokenAmount to be raised

ID	Severity	Location	Status
5	Low	Home.sol	Ignore

Description

The contract first transfers USDT directly into the pair and then uses router.getAmountsIn(amount, [token, usdt]) to calculate the required tokenAmount, without any upper limit check.

Before or after the transaction is mined, MEVs/bots can change the pool reserve by prepending or clamping this transaction (or triggering sync() first), causing getAmountsIn to return a larger tokenAmount, forcing the user to pay more tokens than expected (the contract does not impose a limit, so the transaction will still succeed).

It is recommended to modify stake(amount, orderType, maxTokenAmount) and require(tokenAmount <= maxTokenAmount).

```
function stake(uint256 amount, OrderType orderType) external {
    require(orderType != OrderType.None, "order type must be not None");
    require(amount >= 50e18, "amount must be greater than 50");
    OrderUser storage user = orderUserOf[msg.sender];

    require(user.orderIds.length < 10, "order count must be less than 10");
    require(
        user.accSupply + amount * 2 <= IHome(home).allSupply(msg.sender),
        "exceed max supply"
    );

    IERC20(usdt).safeTransferFrom(msg.sender, pair, amount);
    address[] memory path = new address[](2);
    path[0] = address(token);
    path[1] = address(usdt);
    uint256 tokenAmount = router.getAmountsIn(amount, path)[0];
    IERC20(token).safeTransferFrom(msg.sender, address(this), tokenAmount);
    IERC20(token).safeTransfer(pair, tokenAmount);
    IPair(pair).sync();
    uint256 orderId = orders.length;
    orders.push(Order(orderType, msg.sender, amount * 2, block.timestamp));

    user.orderIds.push(orderId);
    user.accSupply += amount * 2;
    emit Stake(msg.sender, amount * 2, orderType);
```

Recommendation

It is recommended to modify stake(amount, orderType, maxTokenAmount) and require(tokenAmount <= maxTokenAmount).

Status

Ignore.Customer response is logical.

4.2.6 Privileged roles can modify multiple variables in the contract

ID	Severity	Location	Status
6	Low	Home.sol	Fixed

Description

Currently, project privileged roles can modify multiple key variables, such as token transfer whitelists; Achievement updates account levels; NodeFarm sets nodeRobot and achievement, etc. It is recommended that privileged roles use multi-signature management to avoid single point problems.

```
modifier onlyAdmin() {
    require(
        _permissionAdmin.hasRole(DEFAULT_ADMIN_ROLE, msg.sender),
        "PermissionControl: caller does not have admin role"
    );
    _;
}

modifier onlyManager() {
    require(
        _permissionAdmin.hasRole(MANAGER_ROLE, msg.sender),
        "PermissionControl: caller does not have manager role"
    );
    _;
}
```

Status

Fixed.The customer will give up the permission after confirming that the online system is stable.

4.2.7 lastIndex variable is not used

ID	Severity	Location	Status
7	Informational	claim.sol	Fixed

Description

lastIndex is calculated and updated in the loop, but the variable is not used at the end of the function, as if it were intended to advance the index.

```
function claim() external {
    uint256 startIndex = userLastIndexOf[msg.sender];
    uint256 length = userCyclesOf[msg.sender].length;
    uint256 diff = length - startIndex;
    if (diff > 30) {
        diff = 30;
    }
    uint256 reward = 0;
    uint256 lastIndex;
    uint256 lastCycle;
    for (uint256 i = startIndex; i < startIndex + diff; i++) {
        lastCycle = userCyclesOf[msg.sender][i];
        if (currentCycle > lastCycle) {
            reward += perUserRewardOf[lastCycle];
            lastIndex = i;
            emit Claim(msg.sender, perUserRewardOf[lastCycle], lastCycle);
        }
    }
    userLastIndexOf[msg.sender] += diff;
```

Recommendation

It is recommended to confirm the role of lastIndex in the loop and delete it if it is useless.

Status

Fixed.

4.2.8 No event log for key methods

ID	Severity	Location	Status
8	Informational	Achievement.sol	Ignore

Description

The upLevelByNodeFarm and upLevelByManager entries can bypass the threshold check and directly upgrade the account level to any newLevel. Currently, no event is recorded.

Additionally, you need to add events in the following locations:

Family._makeRelationFrom()

Home.changeLevelPool()

Interaction.claim()

```
function _upgradeToLevel(
    uint8 origin,
    uint8 current,
    address account
) internal {
    require(current > origin, "unable to upgrade");

    infoOf[account].level = current;
    // TODO: switch share pool

    IHome(home).changeLevelPool(account, origin, current);
}
```

Recommendation

Suggest adding events to key methods.

Status

Ignore.Customer response is logical.

5 Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Gas Optimization

Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a struct assignment operation affecting an in-memory struct rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.

Coding Style

Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as constant contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Disclaimer

This report is issued in response to facts that occurred or existed prior to the issuance of this report, and liability is assumed only on that basis. Shield Security cannot determine the security status of this program and assumes no responsibility for facts occurring or existing after the date of this report. The security audit analysis and other content in this report is based on documents and materials provided to Shield Security by the information provider through the date of the insurance report. in Shield Security's opinion. The information provided is not missing, falsified, deleted or concealed. If the information provided is missing, altered, deleted, concealed or not in accordance with the actual circumstances, Shield Security shall not be liable for any loss or adverse effect resulting therefrom. shield Security will only carry out the agreed security audit of the security status of the project and issue this report. shield Security is not responsible for the background and other circumstances of the project. Shield Security is not responsible for the background and other circumstances of the project.