Gana Audit Report

Executive Summary

On Nov 28, 2025, the Gana team engaged Audit911 to conduct a thorough security audit of their project. The primary objective was identifying and mitigating potential security vulnerabilities, risks, and coding issues to enhance the project's robustness and reliability. Audit911 conducted this assessment over 2 person days, involving 2 engineers who reviewed the code over a span of 2 day. Employing a multifaceted approach that included static analysis, fuzz testing, formal verification, and manual code review, the Audit911 team identified no issues across different severity levels and categories.

Scope

Project Name	Gana
Staking contract	https://bscscan.com/address/0x10786448c23acb371522577008078851df9fb502#code
Gana contract	https://bscscan.com/address/0x458eab90f190819dbf7cb09c5da2f1e5e4319b2a#writeContract
Fixed	https://bscscan.com/address/0x8773af45b12e8125ed86ffa07cdc875824815989 https://bscscan.com/address/0x72212f35ac448fe7763aa1bfdb360193fa098e52
Language	Solidity

Disclaimer

The audit does not ensure that it has identified every security issue in the smart contracts, and it should not be seen as a confirmation that there are no more vulnerabilities. The audit is not exhaustive, and we recommend further independent audits and setting up a public bug bounty program for enhanced security verification of the smart contracts. Additionally, this report should not be interpreted as personal financial advice or recommendations.

Auditing Process

- Static Analysis: We perform static analysis using our internal tools and Slither to identify potential vulnerabilities and coding issues.

- Fuzz Testing: We execute fuzz testing with our internal fuzzers to uncover potential bugs and logic flaws.
- Invariant Development: We convert the project into Foundry project and develop Foundry invariant tests for the project based on the code semantics and documentations.
- Invariant Testing: We run multiple fuzz testing tools, including Foundry , to identify violations of invariants we developed.
- Formal Verification: We develop individual tests for critical functions and leverage Halmos to prove the functions in question are not vulnerable.
- Manual Code Review: Our engineers manually review code to identify potential vulnerabilities not captured by previous methods.

Vulnerability Severity

We divide severity into three distinct levels: high, medium, low. This classification helps prioritize the issues identified during the audit based on their potential impact and urgency.

- **High Severity Issues** represent critical vulnerabilities or flaws that pose a significant risk to the system's security, functionality, or performance. These issues can lead to severe consequences such as fund loss, or major service disruptions if not addressed immediately. High severity issues typically require urgent attention and prompt remediation to mitigate potential damage and ensure the system's integrity and reliability.
- **Medium Severity Issues** are significant but not critical vulnerabilities or flaws that can impact the system's security, functionality, or performance. These issues might not pose an immediate threat but have the potential to cause considerable harm if left unaddressed over time. Addressing medium severity issues is important to maintain the overall health and efficiency of the system, though they do not require the same level of urgency as high severity issues.
- **Low Severity Issues** are minor vulnerabilities or flaws that have a limited impact on the system's security, functionality, or performance. These issues generally do not pose a significant risk and can be addressed in the regular maintenance cycle. While low severity issues are not critical, resolving them can help improve the system's overall quality and user experience by preventing the accumulation of minor problems over time.

Below is a summary of the vulnerabilities with their current status, highlighting the number of issues identified in each severity category and their resolution progress.

	Number	Resolved
High Severity Issues	1	1
Medium Severity Issues	4	4
Low Severity Issues	0	0
Info Severity Issues	0	0

Findings

[Med]The `buyToken` function has an incorrect parameter.

When executing `swapTokensForExactTokens`, the target address (to) for receiving tokens is hardcoded as `_otherDevAddress` instead of the caller `_msgSender()`. When a user calls this function to buy tokens, the user pays USDT, but the tokens are sent to the project/developer's wallet. The user not only doesn't receive the tokens but also loses their principal.

```
function buyToken(uint256 amount) public {
    require(amount > 0, ">0");
    address[] memory backPath = new address[](2);
    backPath[0] = USDTs;
    backPath[1] = address(this);

    uint256[] memory amountsIn = uniswapV2Router.getAmountsIn(amount, backPath);
    uint256 amountUSDT = amountsIn[0];
    _buyUsdtAmount[_msgSender()] += amountUSDT;
    IERC20(USDTs).transferFrom(_msgSender(),address(this), amountUSDT);
    uniswapV2Router.swapTokensForExactTokens(
        amount,
        amountUSDT,
        backPath,
        address(_otherDevAddress),// ←——
        block.timestamp
    );
}
```

Status: knowledge,The current business scenario is that when a user wants to withdraw a coin within the system, they need to call the `buyToken` function to purchase a coin and deposit it into our address. Then, after 24 hours, the coin will be given to them, so the buyToken will be deposited into one of our addresses.

[Med]Selling in batches cleared out costs.

This only affects users who sell in batches.

Assume a user holds 1000 tokens with a total cost of 100U.

1. Price doubles: 1000 tokens are now worth 200U.
2. Phased profit-taking: The user sells 600 tokens, worth 120U.
3. Trigger logic: 120U (`usdtAmount`) > 100U (`buyUsdtAmount`).
4. Execution result: The user's `_buyUsdtAmount` is forcibly reset to `0`.
5. Disastrous consequences: The user still has 400 tokens (worth 80U). However, the system records their cost as 0. The next time the user sells these 400 tokens, the system calculates the profit: (80 - 0) = 80. The system determines this 80U is all profit, triggering the maximum 25% profit tax.

```
// If the USDAmount obtained from this sale is greater than the total historical purchase cost (buyUSDAmou
nt)
if (usdtAmount > buyUsdtAmount) {
    _buyUsdtAmount[sender] = 0; // ←—— [Fatal Error] Directly reset the cost of all remaining tokens to zer
o.
```

```
    // ...
}
```

Comparison Items	First Sale (Value 120 USDT)	Second Sale (Value 80 USDT)
Cost Assessed	100 USDT	0 USDT
Profit Assessed	20 USDT	80 USDT
Profit Margin	16.6% (20/120)	100% (80/80)
Actual Tax Paid	Approximately 5 USDT (25 coins)	20 USDT (100 coins)
Actual Overall Tax Rate	4.1%	25% (Maximum)

Status: knowledge.The cost of profit tax is not calculated at 25%, but an additional 25% tax will be levied on the profit portion.

Trigger logic: 120U (usdtAmount) > 100U (buyUsdtAmount). At this point, the profit tax will be calculated on the remaining amount because the initial investment has been recouped.

[Med]Cost of using Skim for slip-free cleaning

The tax rate (25%) is calculated based on the user's purchase cost (`_buyUsdtAmount`). If a user normally "sells and buys back" on Uniswap to increase their cost, it will result in significant slippage (price impact) and a 0.3% transaction fee from Uniswap. However, by directly transferring funds to a Pair and then skimming back, the contract's "buy/sell" logic can be triggered without changing the coin price, thus "laundering" profits at a very low cost. For example, a hacker (or large investor) initially spent 100 USDT to buy coins, which are now worth 10,000 USDT. If they sell directly, the huge profit will trigger a 25% profit tax (`_sellProfitFee`).

To evade the tax, the hacker can do the following:

Step 1: Directly transfer funds to a Pair (faking a sell). The hacker calls `token.transfer(pairAddress, amount)`.

Contract reaction: `_autoPair[to]` is true, indicating a sell.

Consequence: A 5% sales tax is deducted. Because this is a direct transfer, the hacker hasn't received the USDT yet. So, although profit calculation is triggered, since no actual swap occurred, the hacker only loses 5% of the coins. (Note: If this step triggers a high profit tax, the hacker might operate in batches or before the coin price becomes extremely volatile).

Step Two: Calling the Pair's Skim function (faking a buy). At this point, the Pair contract contains the coins the hacker just transferred (after tax). Since no swap was called, Uniswap's reserves remain unchanged. The hacker calls `IUniswapV2Pair(pairAddress).skim(hackerAddress)`.

Pair's reaction: Transfers the excess coins back to the hacker.

Contract reaction: `_transfer` is triggered, from is Pair, `_autoPair[from]` is `true`, indicating a buy.

```
// Triggering buy logic
address[] memory path = new address[](2);
path[0] = USDTs;
path[1] = address(this);
//The contract calculates the current value of these coins in USDT.
uint[] memory amounts = uniswapV2Router.getAmountsIn(amount, path);
```

```
// [Core Vulnerability] The cost for hackers has been updated to the current high price!!
_buyUsdtAmount[tx.origin] += amounts[0];
```

Initial State:

- Hacker's Holdings: 10,000 GANA
- Hacker's Cost (`_buyUsdtAmount`): **100 U** (early, extremely low cost purchase)
- Current Market Value: **10,000 U**
- *If sold normally now, the profit would be 9,900 U, with a 25% tax.*

Step 1: Fake Sell (Transfer to Pair)

- The hacker transfers 10,000 GANA to Pair.
- **Contract Logic**: This is considered a sell.
- **Tax Deduction**: 5% (500 GANA) is deducted, so Pair actually receives 9,500 GANA.
- **Cost Change**: Because market value (10,000) > cost (100), `_buyUsdtAmount` is reset to **0**.
- **Asset Change**: The hacker no longer holds GANA and has not received USDT. All GANA is now held by Pair.

Step Two: Fake Buy (Call Skim)

- The hacker calls `Pair.skim(hacker's address)`.
- `Pair` discovers it has received an extra 9,500 GANA (Balance > Reserve), so it returns these 9,500 to the hacker.
- **Contract Logic**:
- Detects `from` is `Pair`.
- Determines it as a **"buy"**.
- **Value Calculation**: `uniswapV2Router.getAmountsIn(9500 GANA)`.
- Assuming the current price remains unchanged, these 9,500 GANA are worth approximately **9,500 U**.
- **Cost Change**:

`_buyUsdtAmount[tx.origin] += amounts[0]; // 0 + 9500 = 9500`

The hacker's cost changes from 0 to 9,500 U! **Step 3: Real Sell (Swap)

- The hacker now holds 9,500 GANA.
- The recorded cost is 9,500 USDT.
- The hacker sells it normally on Uniswap.
- **Contract Judgment**:
- Selling value ≈ 9,500 USDT.
- Recorded cost ≈ 9,500 USDT.
- **Profit ≈ 0**.
- **Result**: Successfully avoided the 25% profit tax.

Status: Fixed.

[High]The return in the burn function is always 1% of the principal.

```
reward = calcItem(user_record); // This calculates the sum of [principal + interest].

uint256 maxInterest = (amount * 45) / 100; // This calculation represents 45% of the principal.

// Logical judgment: If (principal + interest) > (principal * 0.45)if (reward > maxInterest) {
   reward =  (amount * 1) / 100;
}
```

The reward includes the principal. As long as the user doesn't lose more than 55% of their principal, (principal + interest) will always be greater than (principal * 0.45). This if condition is triggered 100% of the time for normal users. For example, if a user deposits 100 USDT, upon redemption at maturity, the code checks that 100 + interest > 45, and therefore executes `reward = 1`.

Status: Fixed.

[Med] The logic for adding a "37% fee" during withdrawal (Unstake).

Current Code Logic (Problem): Assume a user's withdrawal generates 100 USDT in pure interest. With the current configuration (25% repurchase, 5% market, 37% new issuance):

Step 1: The contract sells the user's staked tokens, receiving 100 USDT in return.

Step 2: Deduct the repurchase fee of 25 USDT and the market fee of 5 USDT.

Step 3 (Conflict Point): When distributing funds to the user, only the first two deductions are made. User's actual return: 100 - 25 - 5 = 70 USDT.

At this point, 33% of the 100 USDT remains undistributed.

However, the code then needs to pay 37 USDT to the "37% address".

Result: Total payment required: 25 + 5 + 70 + 37 = 137 USDT.

Funding gap: 37 USDT.

Current Code's Brute-Force Solution: To fill this gap, the contract will initiate a second sell-off, forcibly selling another batch of tokens to reach the 37 USDT requirement.

status: Our business requires 37% to be allocated to the community, hence the specific business requirements.

Status: knowledge.Our business requires 37% to be allocated to the community, hence the specific business requirements.