Gana审计报告

概要

2025年11月30日，Gana团队委托Audit911对其项目进行全面的安全审计。主要目标是识别并缓解潜在的安全漏洞、风险和编码问题，以增强项目的稳健性和可靠性。Audit911团队历时两天完成了此次评估，由两名工程师在两天内对代码进行了审查。Audit911团队采用了包括静态分析、模糊测试、形式化验证和人工代码审查在内的多方面方法，最终未发现任何不同严重级别和类别的问题。

范围

项目名	Gana
staking合约	https://bscscan.com/address/0x10786448c23acb371522577008078851df9fb502
Gana合约	https://bscscan.com/address/0x458eab90f190819dbf7cb09c5da2f1e5e4319b2a
修复	https://bscscan.com/address/0x8773af45b12e8125ed86ffa07cdc875824815989 https://bscscan.com/address/0x72212f35ac448fe7763aa1bfdb360193fa098e52
语言	Solidity

免责声明

本次审计并不能保证已识别出智能合约中的所有安全问题，也不应被视为确认不存在其他漏洞。本次审计并非详尽无遗，我们建议进行进一步的独立审计，并建立公开的漏洞赏金计划，以加强对智能合约的安全性验证。此外，本报告不应被解释为个人财务建议或推荐。

审计流程

- 静态分析：我们使用内部工具和 Slither 进行静态分析，以识别潜在的漏洞和代码问题。
- 模糊测试：我们使用内部模糊测试工具执行模糊测试，以发现潜在的错误和逻辑缺陷。
- 不变式开发：我们将项目转换为 Foundry 项目，并根据代码语义和文档为项目开发 Foundry 不变式测试。
- 不变式测试：我们运行包括 Foundry 在内的多种模糊测试工具，以识别违反我们开发的不变式的情况。
- 形式化验证：我们针对关键功能开发单独的测试用例，并利用 Halmos 来验证相关功能是否存在漏洞。
- 人工代码审查：我们的工程师会人工审查代码，以识别先前方法未能发现的潜在漏洞。

漏洞严重性

我们将严重性分为三个不同的级别：高、中、低。这种分类有助于根据潜在影响和紧迫性对审计过程中发现的问题进行优先级排序。

- **高严重性问题** 指的是对系统安全、功能或性能构成重大风险的关键漏洞或缺陷。如果不立即解决，这些问题可能会导致严重的后果，例如资金损失或重大服务中断。高严重性问题通常需要紧急关注和及时修复，以减轻潜在损害并确保系统的完整性和可靠性。
- **中严重性问题** 指的是可能影响系统安全、功能或性能的重要但不关键的漏洞或缺陷。这些问题可能不会立即构成威胁，但如果不及时解决，可能会造成相当大的损害。解决中等严重性问题对于维护系统的整体健康和效率至关重要，尽管它们不像高严重性问题那样需要紧急处理。
- **低严重性问题**是指对系统安全性、功能或性能影响有限的轻微漏洞或缺陷。这些问题通常不会构成重大风险，可以在常规维护周期中解决。虽然低严重性问题并非关键问题，但解决它们有助于提高系统的整体质量和用户体验，防止小问题随着时间的推移而累积。

以下是漏洞及其当前状态的摘要，重点列出了每个严重性类别中已识别的问题数量及其解决进度。

	数量	已解决
高危问题	1	1
中危问题	4	4
低危问题	0	0
信息类危问题	0	0

发现

`buyToken` 函数的参数不正确

执行 `swapTokensForExactTokens` 时，接收代币的目标地址（to）被硬编码为 `_otherDevAddress`，而不是调用者的 `_msgSender()` 地址。当用户调用此函数购买代币时，用户支付了 USDT，但代币却被发送到了项目/开发者的钱包。用户不仅收不到代币，而且本金也损失了。

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

状态：已知。目前的业务场景是，当用户想要在系统内提现代币时，他们需要调用 buyToken 函数购买代币并将其存入我们的地址。24 小时后，代币将发放给用户，因此 buyToken 将被存入我们的某个地址。

批量出售可以清除成本

此操作仅影响批量出售的用户。

假设用户持有 1000 个代币，总成本为 100U。

1. 价格翻倍：1000 个代币现在价值 200U。
2. 分阶段获利了结：用户出售 600 个代币，价值 120U。
3. 触发逻辑：120U (`usdtAmount`) > 100U (`buyUsdtAmount`)。
4. 执行结果：用户的 `_buyUsdtAmount` 被强制重置为 `0`。
5. 灾难性后果：用户仍然持有 400 个代币（价值 80U）。然而，系统将它们的成本记录为 0。下次用户出售这 400 个代币时，系统计算利润：（80 - 0）= 80。系统认定这 80U 全部为利润，从而触发最高 25% 的利润税。

```
// If the USDAmount obtained from this sale is greater than the total historical purcha
se cost (buyUSDAmount)
if (usdtAmount > buyUsdtAmount) {
    _buyUsdtAmount[sender] = 0; // ←—— [Fatal Error] Directly reset the cost of all re
maining tokens to zero.
    // ...
}
```

状态：已知。利润税并非按 25% 计算，而是对利润部分额外征收 25% 的税。触发逻辑：120U（美元金额）> 100U（买入美元金额）。此时，由于初始投资已收回，利润税将基于剩余金额计算。

使用 Skim 进行无滑点洗钱的成本

税率 (25%) 基于用户的购买成本 (`_buyUsdtAmount`) 计算。如果用户通常在 Uniswap 上“卖出再买回”来增加成本，这将导致显著的滑点（价格冲击）以及 Uniswap 收取的 0.3% 交易费。然而，通过直接将资金转入交易对，然后使用 Skim 进行回购，可以在不改变币价的情况下触发合约的“买卖”逻辑，从而以极低的成本“洗钱”。例如，黑客（或大投资者）最初花费 100 USDT 购买代币，现在这些代币价值 10,000 USDT。如果他们直接出售，巨额利润将触发 25% 的利润税 (`_sellProfitFee`)。

为了逃避税款，黑客可以采取以下步骤：

步骤 1：直接向交易对转账（伪造卖出）。黑客调用 `token.transfer(pairAddress, amount)` 。

合约响应： `_autoPair[to]` 为真，表示已卖出。

结果：扣除 5% 的销售税。由于这是直接转账，黑客尚未收到 USDT。因此，尽管触发了利润计算，但由于没有实际的兑换操作，黑客只会损失 5% 的代币。（注意：如果此步骤触发了高额利润税，黑客可能会分批操作，或者在代币价格剧烈波动之前进行。）

步骤 2：调用交易对的 Skim 函数（伪造买入）。此时，交易对合约包含黑客刚刚转账的代币（已扣除税款）。由于没有进行兑换操作，Uniswap 的储备保持不变。黑客调用 `IUniswapV2Pair(pairAddress).skim(hackerAddress)` 。

交易对的反应：将多余的代币转回给黑客。

合约反应：触发 `_transfer` 函数，源地址为交易对， `_autoPair[from]` 为 `true` ，表示买入操作。

```
// 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 pric
e!!
_buyUsdtAmount[tx.origin] += amounts[0];
```

状态：已修复。

销毁函数的回报始终为本金的 1%

```
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 > maxIntere
st) {
    reward =  (amount * 1) / 100;
}
```

奖励包含本金。只要用户损失的本金不超过 55%，（本金 + 利息）始终大于（本金 * 0.45）。对于普通用户，此条件 100% 会触发。例如，如果用户存入 100 USDT，到期赎回时，代码会检查 100 + 利息 > 45，因此执行 `reward = 1`。

状态：已修复。

提现（解除质押）时添加“37% 手续费”的逻辑

当前代码逻辑（问题）：假设用户提现产生 100 USDT 的纯利息。在当前配置下（25% 回购费，5% 市价费，37% 新发行）：

步骤 1：合约出售用户质押的代币，获得 100 USDT 作为回报。

步骤 2：扣除 25 USDT 的回购费和 5 USDT 的市价费。

步骤 3（冲突点）：在向用户分配资金时，仅扣除前两项费用。用户实际收益：100 - 25 - 5 = 70 USDT。

此时，100 USDT 中有 33% 未分配。

然而，代码还需要向“37% 地址”支付 37 USDT。

结果：所需支付总额：25 + 5 + 70 + 37 = 137 USDT。

资金缺口：37 USDT。

当前代码的暴力破解方案：为了填补这一缺口，合约将发起第二次抛售，强制出售另一批代币以达到 37 USDT 的要求。

状态：我们的业务要求将 37% 的资金分配给社区，因此有这些具体的业务需求。

状态：已知。我们的业务要求将 37% 的资金分配给社区，因此有这些具体的业务需求。