TagAI

Audit Report

Sat Mar 15 2025

contact@bitslab.xyz https://twitter.com/scalebit_

ScaleBit

TagAI Audit Report

1 Executive Summary

1.1 Project Information

Description	A Pump Token Contract
Type	Pump
Auditors	ScaleBit
Timeline	Wed Mar 12 2025 - Sat Mar 15 2025
Languages	Solidity
Platform	BSC
Methods	Architecture Review, Unit Testing, Manual Review

1.2 Files in Scope

The following are the SHA1 hashes of the original reviewed files.

ID	File	SHA-1 Hash
PUM	Pump.sol	6bafa5356a15c88b773a3025cb72c78b2f16b021
TOK	Token.sol	a1f1347576af3acd231dab7448ebe5ae6b6a2a93

1.3 Issue Statistic

Item	Count	Fixed	Acknowledged
Total	7	7	0
Informational	2	2	0
Minor	3	3	0
Medium	2	2	0
Major	0	0	0
Critical	0	0	0

1.4 ScaleBit Audit Breakdown

ScaleBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Integer overflow/underflow
- Number of rounding errors
- Unchecked External Call
- Unchecked CALL Return Values
- Functionality Checks
- Reentrancy
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic issues
- Gas usage
- Fallback function usage
- tx.origin authentication
- Replay attacks
- Coding style issues

1.5 Methodology

The security team adopted the **"Testing and Automated Analysis"**, **"Code Review"** and **"Formal Verification"** strategy to perform a complete security test on the code in a way that is closest to the real attack. The main entrance and scope of security testing are stated in the conventions in the "Audit Objective", which can expand to contexts beyond the scope according to the actual testing needs. The main types of this security audit include:

(1) Testing and Automated Analysis

Items to check: state consistency / failure rollback / unit testing / value overflows / parameter verification / unhandled errors / boundary checking / coding specifications.

(2) Code Review

The code scope is illustrated in section 1.2.

(3) Audit Process

- Carry out relevant security tests on the testnet or the mainnet;
- If there are any questions during the audit process, communicate with the code owner in time. The code owners should actively cooperate (this might include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- The necessary information during the audit process will be well documented for both the audit team and the code owner in a timely manner.

2 Summary

This report has been commissioned by TagAI to identify any potential issues and vulnerabilities in the source code of the Pump smart contract, as well as any contract dependencies that were not part of an officially recognized library. In this audit, we have utilized various techniques, including manual code review and static analysis, to identify potential vulnerabilities and security issues.

During the audit, we identified 7 issues of varying severity, listed below.

ID	Title	Severity	Status
PUM-1	Single-step Ownership Transfer Can be Dangerous	Medium	Fixed
PUM-2	Missing Chain id in the Hash Data	Medium	Fixed
PUM-3	Signature Malleability	Minor	Fixed
PUM-4	Unused Imports	Informational	Fixed
TOK-1	The `approve` Operation is Useless	Minor	Fixed
TOK-2	Fee Evasion	Minor	Fixed
TOK-3	Unused Storage Variable	Informational	Fixed

3 Participant Process

Here are the relevant actors with their respective abilities within the Pump Smart Contract :
Admin

- `adminChangeIPShare` : Updates the IP share address for the platform.
- `adminChangeCreateFee` : Adjusts the fee required to create a token.
- `adminChangeFeeRatio` : Modifies the fee ratio.
- `adminChangeClaimSigner` : Changes the address of the claim signer.
- `adminSetClaimFee` : Sets the fee for claiming rewards.
- `adminChangeFeeAddress` : Updates the address where fees are received.

User

- `createToken` : Allows users to create a new token by paying a fee.
- `userClaim` : Lets a user claim a reward by verifying the order and signature.
- `buyToken` : Allows users to purchase tokens using a bonding curve pricing mechanism.
- `sellToken` : Enables users to sell tokens based on the bonding curve price, considering slippage and fees.

4 Findings

PUM-1 Single-step Ownership Transfer Can be Dangerous

Severity: Medium

Status: Fixed

Code Location:

Pump.sol#5;

Pump.sol#7

Descriptions:

Single-step ownership transfer means that if a wrong address was passed when transferring ownership or admin rights it can mean that role is lost forever. If the admin permissions are given to the wrong address within this function, it will cause irreparable damage to the contract. Below is the official documentation explanation from OpenZeppelin

https://docs.openzeppelin.com/contracts/4.x/api/access

Ownable is a simpler mechanism with a single owner "role" that can be assigned to a single account. This simpler mechanism can be useful for quick tests but projects with production concerns are likely to outgrow it.

```
import "@openzeppelin/contracts/access/Ownable.sol";
```

Suggestion:

It is recommended to use a two-step ownership transfer pattern.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

PUM-2 Missing Chain id in the Hash Data

Severity: Medium

Status: Fixed

Code Location:

Pump.sol#235

Descriptions:

The `userClaim()` function allows users to claim rewards by verifying a signature. However, the hash data used for generating and verifying the signature does not include the chain ID.

```
bytes32 data = keccak256(abi.encodePacked(token, orderId, msg.sender, amount));
    if (!_check(data, signature)) {
        revert InvalidSignature();
    }
```

This omission creates a vulnerability known as a replay attack, where a valid signature on one chain can be reused on another chain.

Suggestion:

It is recommended to include the Chain ID in the Hash Data.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

PUM-3 Signature Malleability

Severity: Minor

Status: Fixed

Code Location:

Pump.sol#267

Descriptions:

The elliptic curve used in Ethereum for signatures is symmetrical, hence for every [v,r,s] there exists another [v,r,s] that returns the same valid result. Therefore two valid signatures exist. `ecrecover()` is vulnerable to signature malleabilit, so it can be dangerous to use it directly.

```
function _check(bytes32 data, bytes calldata sign) internal view returns (bool) {
    bytes32 r = abi.decode(sign[:32], (bytes32));
    bytes32 s = abi.decode(sign[32:64], (bytes32));
    uint8 v = uint8(sign[64]);
    if (v < 27) {
        if (v == 0 || v == 1) v += 27;
    }
    bytes memory profix = "\x19Ethereum Signed Message:\n32";
    bytes32 info = keccak256(abi.encodePacked(profix, data));
    address addr = ecrecover(info, v, r, s);
    return addr == claimSigner;
}
```

Suggestion:

It is recommended to use OpenZeppelin's ECDSA.sol library.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

PUM-4 Unused Imports

Severity: Informational

Status: Fixed

Code Location:

Pump.sol#4;

Pump.sol#6

Descriptions:

The contract imports the following OpenZeppelin libraries but does not use them:

```
import "@openzeppelin/contracts/proxy/Clones.sol";
import "@openzeppelin/contracts/utils/Nonces.sol";
```

Suggestion:

Remove the unused imports to reduce contract size and improve readability.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

TOK-1 The `approve` Operation is Useless

Severity: Minor

Status: Fixed

Code Location:

Token.sol#215

Descriptions:

In the `_makeLiquidityPool()` function, the protocol first approves the router to spend the current contract's tokens.

```
  function _makeLiquidityPool() private {
      _approve(address(this), IPump(manager).getUniswapV2Router(), liquidityAmount);
```

However, in the subsequent steps, the protocol directly transfers tokens and WETH to the Uniswap V2 pair and calls `mint()`.

```
 uint256 tokenAmount = balanceOf(address(this));
      uint256 ethBalance = address(this).balance;

      _transfer(address(this), pair, tokenAmount);
      (bool success, ) = IPump(manager).getWETH().call{value: ethBalance}
(abi.encodeWithSignature("deposit()"));
      require(success, "ETH to WETH failed");
      ERC20(IPump(manager).getWETH()).transfer(pair, ethBalance);

      IUniswapV2Pair(pair).mint(BlackHole);
```

This makes the `approve` operation appear redundant or unnecessary.

Suggestion:

It is recommended to remove the `approve` operation.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

TOK-2 Fee Evasion

Severity: Minor

Status: Fixed

Code Location:

Token.sol#75-150

Descriptions:

In the `sellToken` function, `sellAmount` is checked as follows:

```
if (sellAmount < 100000000) {
    revert DustIssue();
}
```

However, the `buyToken` function lacks a similar check. If `buyFunds` is too low, precision loss may cause the fee to be zero:

```
uint256 buyFunds = msg.value;
uint256 tiptagFee = (msg.value * feeRatio[0]) / divisor;
uint256 sellsmanFee = (msg.value * feeRatio[1]) / divisor;
```

Suggestion:

Implement a minimum fee threshold to ensure that a small `buyFunds` value does not completely evade fees.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

TOK-3 Unused Storage Variable

Severity: Informational

Status: Fixed

Code Location:

Token.sol#30

Descriptions:

In the token contract, the following storage variable is unused:

```
mapping(uint256 => bool) public claimedOrder;
```

Suggestion:

Remove the unused storage variable to optimize contract storage and reduce unnecessary gas costs.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

Appendix 1

Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.