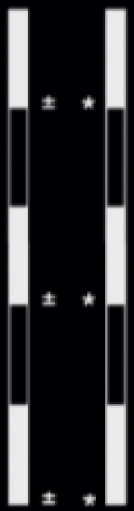
VERILOG X FUZZLAND
JOINT AUDIT

2024

NETZ:
SECURITY REVIEW REPORT



Aug 12, 2024

NETZ Audit Report

Executive Summary

From July 24, 2024, to July 26, 2024, the NETZ project engaged Fuzzland to conduct a thorough security audit of their NETZ project. The primary objective was to identify and mitigate potential security vulnerabilities, risks, and coding issues to enhance the project's robustness and reliability. Fuzzland conducted this assessment over 4 person-days, involving 2 engineers who reviewed the code over a span of 2 days. Employing a multifaceted approach that included static analysis, fuzz testing, and manual code review, Fuzzland team identified 9 issues across different severity levels and categories.

Scope

Project Name	NETZ
Filename	swap-smart-contract-dev-liquidity.zip
Checksum	c396cef9068cc4db86f23c8bdc4204b0
Fix Checksum	f36079a14d50e2c0b043259ddba95b5d
Language	Solidity / Ethereum

Vulnerability Severity

We divide severity into four distinct levels: high, medium, low, and info. 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.
- **Informational Severity Issues** represent informational findings that do not directly impact the system's security, functionality, or performance. These findings are typically observations or recommendations for potential improvements or optimizations. Addressing info severity issues can enhance the system's robustness and efficiency but is not necessary for the system's immediate operation or security. These issues can be considered for future development or enhancement plans.

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	4	4
Medium Severity Issues	0	0
Low Severity Issues	0	0
Informational Severity Issues	5	5

Disclaimer

The audit does not ensure that it has identified every security issue in the project, and it should not be seen as a confirmation that there are no more vulnerabilities. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value projects to commission several independent audits, a public bug bounty program, as well as continuous onchain security auditing and monitoring. Additionally, this report should not be interpreted as personal financial advice or recommendations.

Findings

[High] Incorrect Investor Address Used in Profit Calculation

In the `updateInvestorProfits` function, there is a critical error where the wrong address is used to calculate the investor's eligible investment amount. The function uses `_msgSender()` instead of the `investor` parameter passed to the function.

```
function updateInvestorProfits(address investor, uint256 liquidityCounter)
internal {
    uint256 eligibleInvestmentAmount = getInvestmentAmount(_msgSender());
    uint256 feeShare = (eligibleInvestmentAmount * totalFeesCollectedUSDT) /
liquidityCounter;
    totalFeesCollectedUSDT -= feeShare;

    investors[investor].feeProfits.push(FeeProfit(feeShare, block.number));
}
```

This error leads to investors miscalculating their profit. The function will always calculate profits based on the investment amount of the account calling `updateAllInvestorProfits` (typically the admin or owner), rather than the actual investor's amount. This results in unfair profit distribution, where some investors may receive more profits than they should while others receive less or none at all.

Recommendation:

Replace `_msgSender()` with `investor` in the `getInvestmentAmount` function call:

```
uint256 eligibleInvestmentAmount = getInvestmentAmount(investor);
```

Status: Resolved

[High] Potential Liquidity Calculation Discrepancy in Withdrawal Function

The `withdrawLiquidity` function in the SwapTreasury contract contains a potential discrepancy in calculating the minimum required balance. The function only excludes the admin's balance when calculating the minimum liquidity requirement but does not account for the owner's balance if the owner is different from the admin.

```
if (msg.sender != admin) {
  uint256 minRequiredBalance = (totalLiquidityUSDT -
investorBalancesUSDT[admin]) * minimumLiquidityPercentage / 100;
  require(contractData.usdtContract.balanceOf(address(this)) >=
minRequiredBalance, "Contract is out of balance, try later");
}
```

If the `owner` and `admin` are different addresses, and the `owner` adds liquidity through the `rebalancePool` function, this liquidity will be incorrectly included in the minimum liquidity requirement calculation. This could lead to the contract retaining more liquidity than intended, potentially affecting the withdrawal capabilities of other investors.

Recommendation:

Modify the `withdrawLiquidity` function to exclude both the admin's and owner's balances when calculating the minimum required balance. Alternatively, implement a separate tracking mechanism for administrative liquidity that is not included in the general liquidity calculations.

```
uint256 minRequiredBalance = (totalLiquidityUSDT -
investorBalancesUSDT[admin] - investorBalancesUSDT[owner()]) *
minimumLiquidityPercentage / 100;
```

Status: Resolved

[High] Signature Replay Attack Vulnerability in delegateSwap Function

The `delegateSwap` function is vulnerable to signature replay attacks as it does not implement any mechanism (such as nonce) to prevent the reuse of valid signatures.

```
function delegateSwap(
    bytes memory _signature,
    address _walletAddress,
    uint256 _amount,
    ConversionType _conversionType,
    uint _networkFee,
    address _tokenReceiveAddress
) public onlySigner {
    validateAllowanceAndBalance(_conversionType, _walletAddress, _amount);

    bytes32 message = keccak256(
        abi.encode(_amount, _conversionType, _walletAddress, _networkFee,
_tokenReceiveAddress)
    );
    address signerAddress = getSigner(message, _signature);
    require(signerAddress == _walletAddress, "Invalid user address");
    ...
}
```

A malicious signer could replay a valid signature multiple times, potentially draining user funds or executing unintended swaps.

Recommendation:

Implement a nonce system to prevent signature reuse. Each user should have a nonce that increments with each transaction, and this nonce should be included in the signed message.

```
mapping(address => uint256) public nonces;

function delegateSwap(..., uint256 _nonce) public onlySigner {
    require(_nonce == nonces[_walletAddress], "Invalid nonce");
    nonces[_walletAddress]++;

    bytes32 message = keccak256(abi.encode(_amount, _conversionType,
_walletAddress, _networkFee, _tokenReceiveAddress, _nonce));
    ...
}
```

Status: Resolved

[High] DOS Vulnerability in rebalancePool Function of SwapTreasury Contract

There is a risk of arithmetic underflow in the `rebalancePool` function of the `SwapTreasury` contract, which could cause the function to fail. This issue could prevent the `rebalancePool` function from executing under certain conditions, affecting core contract functionality and liquidity management.

The problematic line exists in the rebalancePool function:

```
uint256 excessLiquidity = amount > totalLiquidityUSDT - contractBalance ?
amount - (totalLiquidityUSDT - contractBalance) : 0;
```

Potential Attack Scenario

1. An attacker or regular user could directly transfer USDT to the contract, making `contractBalance` greater than `totalLiquidityUSDT`.
2. When an admin or owner attempts to call the `rebalancePool` function, the transaction will fail due to arithmetic underflow.
3. This could prevent timely liquidity adjustments, affecting the normal operation of the contract.

Recommendation:

Add checks to ensure `contractBalance` does not exceed `totalLiquidityUSDT`, and implement a handling mechanism if it does.

Status: Resolved

[Info] Missing checks for `address(0)` when assigning values to address state variables

```
File: contracts/SwapTreasury.sol
...
         admin = _admin;
...
     function removeInvestorFromWhitelist(address investor) external
onlyAdminOrOwner {
...
    function addManager(address manager) external onlyOwner {
        managers.push(manager);
        emit ManagerAdded(manager);
    }
```

Recommendation:

Check that the parameter passed in is not adderss(0)

Status: Resolved

[Info] Division by zero not prevented

The divisions below take an input parameter which does not have any zero-value checks, which may lead to the functions reverting when zero is passed.

```
File: contracts/SwapTreasury.sol
...
        uint256 feeShare = (eligibleInvestmentAmount *
totalFeesCollectedUSDT) / liquidityCounter;
...
        uint fees = (((_amount * numerator) / denominator) / 100);
```

Recommendation:

Check that the divisor is not 0 before division

Status: Resolved

[Info] Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

```
File: contracts/SwapTreasury.sol
...
     function _authorizeUpgrade(address newImplementation) internal override
onlyOwner {}
...
     function addManager(address manager) external onlyOwner {

     function removeManager(address manager) external onlyOwner {
...
     function updateAdmin(address newAdmin) external onlyOwner {
...
     ) public onlyOwner {
...
     function updateSubAdmin(address _subAdmin, bool _value) public onlyOwner
{
```

Recommendation:

Ensure that the privileged addresses are multi-sig and/or introduce timelock for improved community oversight. Optionally introduce require statements to limit the scope of the exploits that can be carried out by the privileged addresses.

Status: Resolved

[Info] Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions

See this link for a description of this storage variable. While some contracts may not currently be sub-classed, adding the variable now protects against forgetting to add it in the future.

```
File: contracts/SwapTreasury.sol
...
 import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
...
 import "@openzeppelin/contracts-
upgradeable/proxy/utils/UUPSUpgradeable.sol";
...
 import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
...
 contract SwapTreasury is Initializable, UUPSUpgradeable, OwnableUpgradeable
{
```

Recommendation:

Recommend adding appropriate storage gap at the end of upgradeable contracts.

Status: Resolved

[Info] Not emit an event

Several functions in the code base do not emit relevant events to log their execution.

```
SwapTreasury.updateAllInvestorProfits()
SwapTreasury.updateInvestorProfits(address,uint256)
SwapTreasury.updateEligibilityPeriod(uint256)
SwapTreasury.updateInvestorInvestments(address,uint256)
SwapTreasury.withdrawFees()
SwapTreasury.updateMinimumLiquidityPercentage(uint8)
```

Recommendation:

Consider defining and emitting events whenever sensitive changes occur.

Status: Resolved