Tower Swap Audit Report

Radciper

June 30, 2025

Tower Swap Audit Report

Version 1.0

Radcipher

Audited by: saikumar279

July 9, 2025

Tower Swap Audit Report

Radciper

June 30, 2025

Prepared by: [Radcipher]

Table of Contents

Protocol Summary

TowerSwap merges DeFi and AI to deliver a next-generation launchpad. Gain smarter investment insights powered by artificial intelligence.TowerSwap is transforming the launchpad landscape by fusing AI and DeFi to deliver unmatched intelligence and robust security.

Disclaimer

Radcipher team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Likelihood	Impact		
	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- In Scope: ***Contracts provided by team***

Scope

```
./src/
- IDOConfig.sol
- newIdo.sol
- newIdoFactory.sol
```

Executive Summary

The audit of the TowerSwap smart contracts revealed a generally well-structured DeFi implementation adhering to standard best practices. The code demonstrates a solid foundation with clear attention to transparency and security principles. However, the audit identified 3 high, 2 medium, and 3 low-severity vulnerabilities. While no critical issues were found, the presence of multiple high and medium findings warrants attention. In line with our internal policy, we recommend a follow-up audit, as projects with three or more high or medium vulnerabilities may benefit significantly from additional review after fixes are implemented. TowerSwap has the potential to maintain a strong security posture, provided the identified issues are addressed promptly and thoroughly.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	3
Info	0
Gas	0
Total	8

About

Radcipher is a comprehensive Web3 security firm made up of passionate smart contract researchers, tokenomics experts, and professionals deeply versed in the logic and economic principles behind blockchain protocols. Beyond identifying and addressing critical vulnerabilities in DeFi and NFT projects, we bring a holistic approach combining technical rigor with economic insight. Our team is dedicated to delivering thorough, high-quality security audits that not only focus on bugs but also on the soundness of the underlying protocol logic and tokenomics design. While absolute security can never be guaranteed, we commit to applying our full expertise and meticulous attention to detail to help protect and strengthen your project's foundation. Discover more about our work Radcipher website or connect with us on X @Radcipher.

Disclaimer

No smart contract security review can ensure the complete elimination of vulnerabilities. These reviews are limited by time, resources, and expertise, aiming to identify as many issues as feasible within those constraints. We cannot promise total security following the audit, nor can we guarantee that all problems will be uncovered. We strongly advise implementing ongoing security measures such as additional audits, bug bounty programs, and live on-chain monitoring.

Introduction

Radcipher performed smart contract security assessment of the **Tower Swap** , concentrating on the security aspects of its smart contract implementation.

About Tower Swap

TowerSwap blends artificial intelligence with decentralized finance to create a cutting-edge launchpad platform. Investors benefit from intelligent insights and enhanced security, redefining how projects launch in Web3.

Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact Definitions

- **High** – Results in significant material loss to the protocol or causes substantial harm to a large group of users.
- **Medium** – Causes moderate material loss to the protocol or moderately affects a group of users.
- **Low** – Leads to minor material loss or impacts a small number of users.

Recommended Actions Based on Severity

- **Critical** – Immediate fix required, especially if the contract is already deployed.
- **High** – Must be resolved prior to deployment.
- **Medium** – Recommended to fix.
- **Low / Informational** – Optional fix.

Findings

Disclamer

All reported vulnerabilities were promptly acknowledged and fixed by the team after disclosure.

[H-1] Incorrect Token Amount Calculation for Tokens with Decimals < 18 in contribute() function

Description

In the contribute() function, when calculating tokenAmount for tokens with less than 18 decimals, the final value is not divided by 1e18, leading to precision inflation and incorrect token allocation. Specifically, this branch is problematic:

```
function contribute() external payable onlyNotFinalized {
       uint256 amount = msg.value;

       require(
           block.timestamp >= startTime && block.timestamp <= endTime,
           "Not active"
       );
       require(totalRaised + amount <= hardCap, "Exceeds hard cap");
       require(amount >= minContribution, "Below min");
       require(
           contributors[msg.sender].contribution + amount <= maxContribution,
           "Exceeds max"
       );

       if (contributors[msg.sender].contribution == 0) {
           contributorList.push(msg.sender);
       }
       uint8 tokenDecimals = IERC20Metadata(token).decimals();
       contributors[msg.sender].contribution += amount;
       uint256 tokenAmount;
       if (tokenDecimals == 18) {
            tokenAmount = (amount * rate) / 1e18;
        } else if (tokenDecimals > 18) {
            tokenAmount = (amount * rate * 10 ** (tokenDecimals - 18)) / 1e18;
        } else {
@>            tokenAmount = (amount * rate) / (10 ** (18 - tokenDecimals));
 // @audit issue - not divided by 1e18 precision loss
        }
        contributors[msg.sender].totalTokensAllocated += tokenAmount;

        totalRaised += amount;

        emit Contributed(msg.sender, amount);
    }
```

Impact

This omits a necessary division by 1e18, unlike the other branches. The inconsistency causes users to receive more tokens than intended, resulting in: Over-distribution of tokens Potential depletion of the token pool Fairness violations and economic imbalance

Fix has been su

[H-2] Incorrect Token Amount Calculation for Tokens with Decimals $<$ 18 in _addLiquidity() function

Description

When computing the token amount for liquidity provision, the code path for tokens with decimals less than 18 omits division by 1e18, leading to precision overestimation:

```
function _addLiquidity() internal {
        require(!liquidityAdded, "Liquidity already added");

        // Calculate amounts for liquidity
        uint256 ethAmount = (totalRaised * liquidityPercentage) / 10000;
        uint8 tokenDecimals = IERC20Metadata(token).decimals();

        uint256 tokenAmount;
        if (tokenDecimals == 18) {
            tokenAmount = (ethAmount * rate) / 1e18;
        } else if (tokenDecimals > 18) {
            tokenAmount =
                (ethAmount * rate * 10 ** (tokenDecimals - 18)) /
                1e18;
        } else {
@>            tokenAmount = (ethAmount * rate) / (10 ** (18 - tokenDecimals));
// @audit issue - not divided by 1e18 precision loss
        }

        // Verify contract has sufficient balances
        require(
            IERC20(token).balanceOf(address(this)) >= tokenAmount,
            "Not enough tokens for liquidity"
        );
        require(
            address(this).balance >= ethAmount,
            "Not enough ETH for liquidity"
        );

        // Approve token transfer
        require(
            IERC20(token).approve(dexRouter, tokenAmount),
            "Token approval failed"
        );

        // Add liquidity
        (uint amountToken, uint amountETH, uint liquidity) = IUniswapV2Router(
            dexRouter
        ).addLiquidityETH{value: ethAmount}(
            token,
            tokenAmount,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            address(this), // LP tokens go to this contract
            block.timestamp + 1200 // deadline
        );

        // @audit issue after adding liquidity again
        make the allowance of dexrouter zero

        // Mark liquidity as added
        liquidityAdded = true;

        // Additional events for debugging
        emit LiquidityAdded(amountToken, amountETH, liquidity);
    }
```

Impact

This results in allocating more tokens than intended:

- Excessive token consumption for liquidity
- Failure in liquidity addition due to insufficient balance

This bug affects only tokens with tokenDecimals < 18 and should be corrected by dividing the result by 1e18, maintaining consistency with the other branches.

[H-3] Fee-on-Transfer Tokens Can Break Token Transfer Assumptions in createIDO()

Description

The function assumes that the full requiredTokenAmount will be transferred using transferFrom() and later forwarded using transfer():

```
function createIDO(
        address _token,
        address _dexRouter,
        address _owner,
        IDOConfigLib.Config memory _config
```

```
    ) external returns (address) {
        uint256 requiredTokenAmount = calculateRequiredTokenAmount(
            _config.rate,
            _config.hardCap,
            _config.liquidityPercentage
        );

        // Pastikan factory diapprove
        require(
            IERC20(_token).allowance(msg.sender, address(this)) >= requiredTokenAmount,
            "Factory not approved to spend required tokens"
        );
        // @audit non compatable tokens may be used here like fee on transfer
        which may lead to amount transferred less than requiredTokenAmount
        // Transfer token dari user ke factory
@>        require(
            IERC20(_token).transferFrom(msg.sender, address(this), requiredTokenAmount),
            "Token transfer to factory failed"
        );

        // Deploy IDO contract
        IDO ido = new IDO(
            _token,
            _dexRouter,
            _owner,
            _config
        );

        // Kirim token dari factory ke contract IDO
        require(
            IERC20(_token).transfer(address(ido), requiredTokenAmount),
            "Token transfer to IDO failed"
        );

        address idoAddress = address(ido);
        allIDOs.push(idoAddress);
        emit IDOCreated(idoAddress, _owner);

        return idoAddress;
    }
```

However, fee-on-transfer tokens (like many deflationary tokens) deduct a fee during transfer, meaning: The factory may receive less than requiredTokenAmount The subsequent transfer to the IDO contract may fail Even if it succeeds, the IDO contract will lack sufficient tokens, leading to: Liquidity addition failure Under-allocated contributors Unpredictable or unfair token distribution The protocol currently has no mechanism to detect or reject such tokens during IDO creation.

Mitigation options:

Enforce a whitelist of known ERC20-compliant tokens

[M-1] Missing Validation for IDO Configuration Parameters

Description

The createIDO() function accepts an IDOConfigLib.Config struct that contains critical configuration values for the IDO lifecycle. However, no validation is performed on these parameters before passing them into the IDO contract:

```
IDO ido = new IDO(_token, _dexRouter, _owner, _config);
```

This can lead to misconfigured or malicious IDOs with invalid or harmful parameters, such as: rate == 0 (users receive 0 tokens)

hardCap < softCap (sale can never finalize)

minContribution > maxContribution (no valid contribution possible)

endTime <= startTime (IDO ends before it starts)

liquidityPercentage > 10000 (impossible liquidity allocation)

tgePercentage > 10000 (more than 100% unlocked at TGE)

vestingDuration == 0 while cliffDuration > 0 (conflicting vesting logic)

Impact

Logical failures during IDO execution Funds getting locked or unrecoverable Broken contributor or vesting flows Damaged protocol trust and reputation

[M-2] Token Amount Calculation Ignores Token Decimals in withdrawUnsoldTokens()

Description

In the withdrawUnsoldTokens() function, the calculation of totalTokensForSale and totalTokensForLiquidity directly multiplies totalRaised by rate, without considering the token's decimals:

```
function withdrawUnsoldTokens() external onlyOwner {
        require(finalized || cancelled, "Sale not finished");

        if (finalized) {
            // After successful sale, only allow withdrawing
            excess tokens (not allocated to liquidity)

            /// @audit decimals are not taken into consideration here
@>            uint256 totalTokensForSale = totalRaised * rate;
@>            uint256 totalTokensForLiquidity = (totalRaised *
@>                liquidityPercentage *
@>@>                rate) / 10000;
            uint256 totalTokensNeeded = totalTokensForSale +
                totalTokensForLiquidity;

            uint256 currentBalance = IERC20(token).balanceOf(address(this));
            if (currentBalance > totalTokensNeeded) {
@>                uint256 excess = currentBalance - totalTokensNeeded;
                IERC20(token).transfer(owner, excess);
            }
        } else {
            // After cancelled sale, allow withdrawing all tokens
            uint256 remaining = IERC20(token).balanceOf(address(this));
            IERC20(token).transfer(owner, remaining);
        }
    }
```

Impact

This assumes that the token has 18 decimals, but if the token has fewer or more decimals (e.g., 6 or 8), the calculated values will be inaccurate, leading to: Underestimation or overestimation of required tokens Incorrect excess token calculation Potentially locking up tokens that shouldn't be, or leaking tokens that are still needed

[L-1] Unreachable else Block Due to Redundant Condition in finalizeSale() function

Description

In the finalizeSale() function, the require(totalRaised >= softCap) statement ensures that the soft cap must be reached before proceeding. However, immediately after that, the following conditional is used:

```
if (totalRaised >= softCap) { ... } else { ... }
```

Since the require statement already enforces totalRaised >= softCap, the else block can never be executed, rendering it dead code:

```
function finalizeSale() external onlyOwner onlyNotFinalized {
        require(totalRaised >= softCap, "Softcap not reached");

        if (totalRaised >= softCap) {
           .............
        } else {
            // Sale failed - mark as cancelled
            cancelled = true; // @audit dead code because if condition is same
            as the require condition above which leads else block to never execute
            emit Cancelled();
        }
    }
```

Impact

This dead code: Adds unnecessary complexity and confusion.

Mitigation

Suggests that the contract might have previously supported soft cap failure handling here, which is now redundant.

[L-2] DEX Router Token Allowance Not Reset After Use in _addLiquidity()

Description

After approving the DEX router to spend tokenAmount for liquidity provisioning, the allowance is not reset to zero after the operation:

```
function _addLiquidity() internal {
        require(!liquidityAdded, "Liquidity already added");

        ...............

        // Approve token transfer
@>        require(
            IERC20(token).approve(dexRouter, tokenAmount),
            "Token approval failed"
        );

        // Add liquidity
        (uint amountToken, uint amountETH, uint liquidity) = IUniswapV2Router(
            dexRouter
        ).addLiquidityETH{value: ethAmount}(
            token,
            tokenAmount,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            address(this), // LP tokens go to this contract
            block.timestamp + 1200 // deadline
        );

      // @audit issue after adding liquidity again make the allowance of dexrouter zero

        // Mark liquidity as added
        liquidityAdded = true;

        // Additional events for debugging
        emit LiquidityAdded(amountToken, amountETH, liquidity);
    }
```

Impact

This leaves the router with an indefinite token allowance, which: Violates the principle of least privilege Can pose a security risk if the router address is compromised or upgraded

Mitigation

It is a good practice to reset the allowance to zero after successful usage to minimize exposure:

```
IERC20(token).approve(dexRouter, 0);
```

[L-3] Missing Admin-Only Access Control for IDO Creation

Description

The createIDO() function is currently externally callable by anyone, allowing arbitrary users to launch IDOs with any token:

```
function createIDO(...) external returns (address) { ... }
```

Impact

This may be acceptable for open protocols, but in a platform where curation or token whitelisting is important, this lack of restriction is dangerous. It allows: Creation of IDOs with malicious or broken tokens Abuse of the factory to create spam or rug-pull IDOs Bypassing project-level vetting or verification Restricting createIDO() to a platform admin or validator (e.g., with an onlyAdmin modifier) would allow better control over: What tokens are approved When and how IDOs are launched Compliance and due diligence processes