SecuHash audit for the project zapdefi.com on Nov 18, 2024

Complete Security Audit
by SecuHash

Audit date
November 18, 2024

Project audited
Zap Defi

Contact & Socials
zapdefi.com
@zap_defi
@zap_eth

Table of Contents

1. Executive Summary

This security audit report provides a comprehensive analysis of the Solidity smart contract ZAP Token. The contract was subjected to a rigorous assessment to identify potential vulnerabilities, logical errors, and inefficiencies. Our audit has uncovered several issues ranging from high to informational severity levels. Recommendations have been provided to address each finding and enhance the overall security and performance of the contract.

2. Scope of Audit

The audit covers the following aspects:

- Security: Identification of security vulnerabilities within the contract.
- Correctness: Verification that the contract behaves as expected.
- Best Practices: Ensuring the contract follows Solidity and industry best practices.
- Optimization: Suggestions for gas efficiency and performance improvements

3. Methodology

Our audit process includes:

- Manual Code Review: Line-by-line analysis of the code.
- Automated Analysis: Using static analysis tools to detect vulnerabilities.
- Testing: Simulating various scenarios to test the contract's logic.
- Comparison: Ensuring compliance with ERC20 standards and OpenZeppelin implementations.

4. Contract Overview

The Token contract is an ERC20 token with the following features:

- Trading Control: The owner can enable or disable trading.
- Fee Mechanism: Buy and sell fees are applied, and fees are transferred to a designated fee receiver.
- Whitelist System: Certain addresses can be whitelisted to bypass trading restrictions.
- Automated Market Maker Pair Management: Handles liquidity pool pairs for automated market makers.
- Swap and Liquify Mechanism: Converts accumulated fees into ETH.
- Emergency Functions: Allows the owner to extract ETH or tokens in emergency situations.

5. Detailed Findings

5.1 High Severity Issues

1. Unlimited Minting by Owner

Description:
The mint function allows the owner to mint tokens arbitrarily at any time. There is no cap on the total supply or restrictions on when the owner can mint tokens.

Code Snippet:

```
function mint(address to, uint256 amount) external {
    require(msg.sender == presaleContract ||
msg.sender == owner(), "Not authorized");
    _mint(to, amount);
    totalTokensSold += amount;
}
```

Impact:
The owner can inflate the token supply at will, leading to devaluation of the token and loss of trust among token holders. This poses a significant centralization risk and can be exploited maliciously.

Recommendation:
- Implement a Cap: Introduce a maximum total supply that cannot be exceeded.

- Restrict Minting Period: Allow minting only during the initial deployment or presale phase.

- Burn Ownership Privileges: After the initial distribution, consider renouncing minting rights or transferring ownership to a governance contract.

- Transparent Communication: Clearly communicate minting capabilities and policies to token holders.

Team’s respones:
“We’re going to renounce ownership of the token once the presale is completed to disable the minting functionality”

Status:
✅ **Resolved and acknowledged**

2. Owner Can Set Excessive Fees

Description:
The owner has the ability to set the buy and sell fees up to the MAX_FEE of 25% at any time without any delay or consensus.

Code Snippet:

```
uint256 public constant MAX_FEE = 25; // 25%
uint256 public buyFee = 5;
uint256 public sellFee = 5;

function setFees(uint256 _buyFee, uint256 _sellFee)
external onlyOwner {
    require(_buyFee <= MAX_FEE && _sellFee <=
MAX_FEE, "Fees exceed maximum");
    buyFee = _buyFee;
    sellFee = _sellFee;
    emit FeeUpdated("Buy", _buyFee);
    emit FeeUpdated("Sell", _sellFee);
}
```

Impact:
The owner can suddenly increase fees to the maximum limit, effectively taxing transactions at 25%. This can be used to discourage trading, manipulate the market, or extract excessive fees from users.

Recommendation:
- Implement a Timelock: Introduce a delay between when a fee change is announced and when it becomes effective.

- Fee Change Limits: Limit the percentage by which fees can be changed within a certain time frame.

- Governance Mechanism: Require fee changes to be approved through a decentralized governance process.

- Transparent Communication: Notify users in advance about any fee changes.

Team's respones:

“Having the max fee as 25% is enough to discourage sudden changes in fees”

Status:

✅ **Resolved and acknowledged**

5.2 Medium Severity Issues

1. Potential for Swapping Mechanism to Get Stuck

Description:
If an exception occurs during the `swapAndSendFees` process, particularly within the `swapTokensForEth` function, the `swapping` variable may remain set to `true`. This would prevent further swapping and accumulation of fees

Code Snippet:

```
function swapAndSendFees() private {
    uint256 contractTokenBalance =
balanceOf(address(this));
    bool canSwap = contractTokenBalance >=
swapThreshold;

    if (canSwap && !swapping) {
        swapping = true;

        // Swap tokens for ETH
        swapTokensForEth(contractTokenBalance);

        emit
SwapAndTransferFees(contractTokenBalance,
address(this).balance);

        swapping = false;
    }
}
```

Impact:
The contract could become unable to process fee swaps, leading to an accumulation of tokens in the contract and potential disruption of the fee mechanism.

Recommendation:
- Use Try-Catch Blocks: Implement try-catch around external calls to handle exceptions gracefully.

- Ensure State Reset: Use a finally pattern to reset the swapping variable even if an error occurs.

- Add Fallback Mechanism: Provide a way for the owner to reset the swapping variable manually if it gets stuck.

Team's respones:
"Although it's rare that it would fail we determined that the issue won't happen in the code"

Status:
✅ **Resolved and acknowledged**

2. Centralization Risk Due to Owner Privileges

Description:
The owner holds significant control over various aspects of the contract, including minting tokens, setting fees, and extracting ETH. There is no governance or timelock mechanism to check this power.

Impact:
Centralization of control can lead to misuse of privileges, intentional or accidental, resulting in loss of funds or manipulation of the token's market.

Recommendation:
- Introduce Governance: Implement a decentralized governance model where key decisions require consensus.

- Implement Timelocks: Use timelocks for critical functions to allow users to react to changes.

- Limit Owner Powers: After deployment, consider limiting the owner's abilities or distributing control among multiple parties.

Team's respones:
"Since we'll renounce ownership once the presale is completed, the issue is resolved"

Status:
✅ **Resolved and acknowledged**

5.3 Low Severity Issues

1. Lack of Event Emission in Some Setter Functions

Description:
Functions like `setFeeReceiver` and `setSwapThreshold` modify important contract parameters but do not emit events upon execution.

Code Snippet:

```
function setFeeReceiver(address _feeReceiver)
external onlyOwner {
    require(_feeReceiver != address(0), "Invalid fee
receiver");
    feeReceiver = _feeReceiver;
}

function setSwapThreshold(uint256 _swapThreshold)
external onlyOwner {
    require(_swapThreshold > 0, "Threshold must be >
0");
    swapThreshold = _swapThreshold;
}
```

Impact:
Without events, it's harder for off-chain systems and users to track changes, reducing transparency and trust.

Recommendation:
- Emit Events: Add events for these functions to log changes.

Team's respones:
"Events are a minor issue."

Status:
✅ **Resolved and acknowledged**

2. Integer Division Rounding Errors

Description:
In fee calculations, using integer division may lead to rounding down, potentially causing tiny discrepancies in fee amounts.

Code Snippet:

```
feeAmount = amount * sellFee / 100;
```

Impact:
Minor loss of precision in fee calculation, which is generally acceptable but worth noting.

Recommendation:
- Acknowledge in Documentation: Clarify that fees are calculated using integer division and may be rounded down.

- Consider Using a Multiplier: Use a larger base (e.g., basis points) to increase precision if necessary.

Team's respones:
"Acknowledged."

Status:
✅ **Resolved and acknowledged**

3. No Maximum Wallet Limit

Description:
The contract does not enforce a maximum wallet limit, allowing users to hold unlimited tokens.

Impact:
While not inherently a problem, it may allow for large holders to accumulate significant portions of the supply, posing centralization risks.

Recommendation:

- Assess Need for Max Wallet Limit: Determine if implementing a maximum wallet limit aligns with the project's goals.

- Implement if Necessary: Add logic to enforce a maximum balance per wallet if desired.

Team’s respones:
“Since the presale contract will hold many tokens we don’t need this functionality.”

Status:
✅ **Resolved and acknowledged**

5.4 Informational Notes

1. Code Comments and Documentation

Observation:
The contract lacks comprehensive comments and documentation in some areas, which may hinder understanding for future developers or auditors.

Recommendation:
- Enhance Documentation: Add comments explaining the purpose and functionality of complex functions.

- Maintain Code Clarity: Use clear and descriptive variable and function names.

2. Event Consistency

Observation:
Some functions emit events while others do not, leading to inconsistency.

Recommendation:
- Standardize Event Emission: Ensure that all state-changing functions emit appropriate events for consistency and transparency.

3. Magic Numbers and Constants

Observation:
The use of hardcoded values, such as 10000 * 10**18 for swapThreshold, can reduce code readability.

Recommendation:
- Define Constants: Use uint256 constants with descriptive names for such values.

6. Best Practices and Recommendations

Use SafeMath:
Although Solidity 0.8+ has built-in overflow checks, consider using SafeMath libraries for explicitness.

Access Control with OpenZeppelin's Ownable:
Ensure all functions that should be restricted to the owner are using the onlyOwner modifier.

Reentrancy Guard:
Apply the nonReentrant modifier from OpenZeppelin's ReentrancyGuard on functions that involve external calls.

Event Emission:
Emit events for state-changing functions to improve transparency.

Code Comments and Documentation:
Enhance code comments for better maintainability and clarity.

Input Validation:
Add input validation to setter functions to prevent invalid configurations.

Gas Optimization:
Review the contract for gas inefficiencies, such as redundant state variable reads.

7. Conclusion

The Token contract exhibits several issues that need to be addressed to ensure security and compliance with best practices. By implementing the recommended changes, the contract's robustness and reliability will be significantly improved.

The team has implemented the recommended steps and acknowledged those issues to resolve them as properly as mentioned.

In short the project is high quality and has the integrity necessary to make it a success.