SECUREWEI

SECURITY AUDIT REPORT

Prepared by
SECUREWEI

Prepared for
NYXIUMPAD

Don't trust, verify.

Table of Contents

Summary

About Nyxium Token

NyxiumPad is a decentralized launchpad built on the Polygon network, designed to empower Web3 startups with seamless token launches, presales, and community-driven growth. At its core, the Nyxium (NYX) token fuels the ecosystem, enabling staking, governance, and access to platform utilities.

With a fair, community-first approach, NyxiumPad integrates gamified participation (airdrops, quizzes, and badges) to foster engagement while ensuring long-term sustainability through a limited supply of 1 Million NYX token.

Security Assessment

The security audit of the Nyxium Token smart contract was conducted over a two-day period, concluding on July 2, 2025. During the assessment, our team employed comprehensive manual review to rigorously evaluate the contract's codebase. The audit focused on key security areas, including access control mechanisms, token issuance, potential reentrancy risks, and compliance with relevant token standards. All critical components and interactions within the Nyxium Token contract were given special attention to ensure robust security and adherence to best practices.

Over the course of the audit, we identified a total of six security issues:

- 6 **Info** severity vulnerabilities

Scope

The audit focused exclusively on the Nyxium Token smart contract located at https://polygonscan.com/token/0x0652128E0235dDC297F595c08F510821a85041D1#code. Only the on-chain contract code was assessed; no frontend, server infrastructure, or off-chain components were in scope.
The following files were in scope:

- Nyxium.sol

Type: Solidity

Platform: Polygon Network

Disclaimer

The findings, observations, and conclusions detailed in this audit report are the result of a comprehensive and diligent assessment performed by experienced security professionals. This assessment was conducted in accordance with recognized industry standards and best practices, and reflects the professional judgment of the audit team based on the scope, information, systems, and documentation made available at the time of the engagement.

While every effort has been made to identify and communicate relevant vulnerabilities, limitations inherent in any security review must be acknowledged. No security assessment, regardless of its depth or methodology, can guarantee the complete absence of vulnerabilities or provide an absolute assurance of security. The dynamic nature of software development and the evolving threat landscape mean that new risks may emerge and system conditions may change after the completion of our work.

This report should therefore be regarded as a significant contribution to a broader, ongoing security posture. It is intended to inform risk-aware decision making and support responsible security practices, but it is not a substitute for comprehensive internal controls, secure development practices, or continued vigilance.

Severity Criteria

Severity	Description
High	Critical vulnerabilities where core functionalities of the protocol or funds are at significant risk.
Major	Vulnerabilities that may affect certain financial aspects or cause specific functionalities to malfunction.
Low	Issues without direct financial impact but could affect user experience or minor protocol operations.
Info	Suggestions for improvements such as gas efficiencies, adherence to best practices, or optimizations.
Likelihood	**Description**
High	The vulnerability can be exploited with minimal or no specific conditions.
Major	Exploitation requires certain conditions to be met or may not present sufficient incentives to be exploited.
Low	Exploitation demands rare conditions or incurs financial costs for the attacker.
N/A	Not applicable.

Findings Summary

Severity	Likelihood	Title
Info	N/A	Missing Mitigation for ERC-20 “Approve” Race-Condition
Info	N/A	Prevent Transfers and Approvals to the Zero Address
Info	N/A	Leverage *constant* and *immutable* for fixed-state variables
Info	N/A	Use Custom Errors to Reduce Gas Consumption
Info	N/A	Introduce internal helper functions
Info	N/A	Use of Unchecked Blocks for Safe Subtractions

Detailed Findings

Missing Mitigation for ERC-20 “Approve” Race-Condition

Severity: Info
Likelihood: N/A
Target: Nyxium.sol
Status: Acknowledged
ID: 1

Description

The ERC-20 ***approve*** function in this contract unconditionally overwrites the existing allowance:

```
function approve(address spender, uint256 amount) external
returns(bool) {
     _allowances[msg.sender][spender] = amount;
     emit Approval(msg.sender, spender, amount);
     return true;
 }
```

This pattern is susceptible to the well-known “race-condition” or “double-spend” issue: if a token holder attempts to change a spender’s allowance from a non-zero value to another non-zero value, an attacker (the approved spender) can observe the pending transaction and quickly spend the old allowance before the new approval takes effect. The owner’s intention to reduce or increase the allowance can be subverted, resulting in unintended token transfers.

Impact

While this is not a direct security vulnerability that allows arbitrary token theft, it can lead to a spender draining more tokens than the owner intended.

Recommendation

Adopt one or more of the following mitigation strategies to eliminate the race condition:

- Use Safe Increase/Decrease Functions: provide ***increaseAllowance*** and ***decreaseAllowance*** helpers to adjust allowances relative to their current value, as implemented in OpenZeppelin's ERC-20: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/cc94ea49301460aa6bd39edf29b2ab953d1d80b5/contracts/token/ERC20/utils/SafeERC20.sol#L82

- Require Zero-Reset: enforce that an existing allowance must first be set to zero before it can be changed to a new non-zero amount:

```
function approve(address spender, uint256 amount) external
returns(bool) {
    require(amount == 0 || _allowances[msg.sender][spender] == 0,
"Must zero allowance first");
    .
    .
    .
 }
```

Prevent Transfers and Approvals to the Zero Address

Severity: Info
Likelihood: N/A
Target: Nyxium.sol
Status: Acknowledged
ID: 2

Description

The current implementation of Nyxium does not guard against using the zero address (*address(0)*) as a sender and recipient in *transfer*/*transferFrom* or as a spender in *approve*. As a result, callers can inadvertently:

• Send tokens to *address(0)*, effectively burning them without an explicit burn mechanism.
• Grant allowances to *address(0)*, which is a non-existent account and serves no practical purpose.

While these actions do not introduce critical vulnerabilities, they deviate from the ERC-20 general recommendation that transfers and approvals to the zero address should revert.

Impact

• Token holders may accidentally "burn" their own tokens by transferring to *address(0)*.
• Approvals to the zero address accomplish nothing but clutter on-chain state.
• Overall, inconsistent handling of *address(0)* can lead to potential vulnerabilities when implementing *mint*/*burn* functions as common current implementations treat transfers to *address(0)* as burn and transfer from *address(0)* as mint.

Recommendation

Introduce explicit *require* statements to disallow zero-address operations in all relevant functions (*approve*, *transfer*, *transferFrom*):

```
require(recipient != address(0), "ERC20: transfer to the zero
address");
require(sender != address(0), "ERC20: transfer from the zero
address");
```

Leverage *constant* and *immutable* for fixed-state variables

Severity: Info
Likelihood: N/A
Target: Nyxium.sol
Status: Acknowledged
ID: 3

Description

In the Nyxium contract, the following state variables are assigned once and never modified thereafter:

- *_name*
- *_symbol*
- *_decimals*
- *_totalSupply*

Because these values never change, marking them as constant (or immutable if you need constructor assignment) enables the Solidity compiler to embed them directly into bytecode rather than reading from storage at runtime.

Impact

Leveraging constant and immutable declarations will save gas.

Recommendation

Declare constant values with the constant keyword. e.g.:

```
string private constant _name = "Nyxium";
string private constant _symbol = "NYX";
uint8 private constant _decimals = 18;
uint256 private constant _totalSupply = 1e24;
```

Use Custom Errors to Reduce Gas Consumption

Severity: Info
Likelihood: N/A
Target: Nyxium.sol
Status: Acknowledged
ID: 4

Description

The contract currently uses ***require*** statements with literal revert strings ("Low Balance.", "Low Allowance.") to guard against invalid operations. Since Solidity 0.8.4, you can define and use custom errors instead of long revert strings. Custom errors encode only a selector and any parameters, which is significantly more gas-efficient than storing full strings in the bytecode and emitting them at runtime.

Impact

Increased deployment size and higher per-transaction gas costs whenever a revert string is used.

Recommendation

Define custom errors at the top of your contract and replace the ***require*** calls with your custom errors, for example:

```
error LowBalance();
error LowAllowance();
...
require(_balances[msg.sender] >= amount, LowBalance());
...
require(_allowances[sender][msg.sender] >= amount, LowAllowance());
```

Introduce internal helper functions

Severity: Info
Likelihood: N/A
Target: Nyxium.sol
Status: Acknowledged
ID: 5

Description

The current implementation of ***transfer*** and ***transferFrom*** duplicates core logic across multiple public functions. This leads to code repetition, which increases maintenance overhead, makes future enhancements more error-prone, and makes it harder to ensure consistent behaviour across all token operations.

Impact

While this is not a security vulnerability per se, the duplicated logic increases the risk of introducing inconsistencies and leads to larger bytecode size.

Recommendation

Refactor the contract to extract the common logic into internal helper functions, for example:

```
function _transfer(address sender, address recipient, uint256 amount)
internal {
    require(_balances[sender] >= amount, "Low Balance.");
    _balances[sender] -= amount;
    _balances[recipient] += amount;
    emit Transfer(sender, recipient, amount);
}
```

Use of Unchecked Blocks for Safe Subtractions

Severity: Info
Likelihood: N/A
Target: Nyxium.sol
Status: Acknowledged
ID: 6

Description

In the ***transfer*** and ***transferFrom*** functions, the contract performs balance and allowance checks with ***require*** before subtracting tokens or allowance amounts. Since Solidity 0.8.0 reverts on underflow and the pre-check guarantees ***balance >= amount*** (and ***allowance >= amount***), the subsequent subtraction cannot underflow. Wrapping these operations in an ***unchecked*** block will skip the implicit overflow/underflow checks and reduce gas consumption.

Impact

Gas inefficiency: each subtraction currently incurs an implicit overflow/underflow check, increasing transaction costs.

Recommendation

After each ***require*** that validates sufficient balance or allowance, wrap the subtraction operations in an ***unchecked*** block. For example, in ***transfer***:

```
require(_balances[msg.sender] >= amount, "Low Balance.");
unchecked {
    _balances[msg.sender] -= amount;
}
```

And similarly in ***transferFrom*** for both allowance and balance decrements. This change retains functional correctness while reducing gas usage.

Methodology

The audit was conducted using a comprehensive and multi-layered approach to identify security vulnerabilities, logical flaws, and deviations from best practices within the smart contract codebase. The following methodologies were employed:

Manual Code Review

An extensive manual analysis was performed on the smart contract source code. This involved a line-by-line review to assess logic correctness, access control, input validation, gas optimization, and adherence to industry standards. Manual auditing was also used to identify subtle logic bugs, business logic flaws, and edge cases that automated tools may overlook.

Automated Static Analysis

We employed a suite of state-of-the-art static analysis tools to detect known vulnerability patterns and unsafe programming constructs. These tools scanned the contract code for issues.
Findings from static analysis were carefully reviewed and validated to eliminate false positives and ensure contextual relevance.

Threat Modeling and Risk Assessment

We analyzed the smart contracts within the broader context of their intended ecosystem and business logic to identify potential threat vectors. Scenarios such as front-running, oracle manipulation, and privilege escalation were considered to assess the system's robustness against known attack vectors.

About SecureWei

SecureWei is a blockchain security firm dedicated to safeguarding decentralized systems through rigorous smart contract auditing and protocol assessments. Our mission is to empower developers and projects with the confidence that their code is secure, reliable, and aligned with industry best practices.

At SecureWei, we combine deep domain expertise with cutting-edge analysis techniques to uncover hidden vulnerabilities and design flaws before they can be exploited. Our team brings together professionals with backgrounds in cybersecurity, formal verification, and decentralized application development, enabling us to deliver high-impact, actionable security insights.

Why Choose SecureWei?

- Thoroughness: Every line of code is reviewed by experienced auditors to uncover subtle bugs and complex attack vectors.
- Trust: Our reputation is built on integrity, transparency, and consistently high standards.
- Collaboration: We provide detailed reports and work hand-in-hand with teams to remediate issues effectively.
- Future-Focused: We stay ahead of the curve with evolving blockchain standards and security trends.

Contact

To learn more about our services or to request an audit, feel free to reach out:

- Twitter (X): @securewei
- Telegram: @securewei
- Website: securewei.com

SecureWei is committed to raising the security bar in Web3.