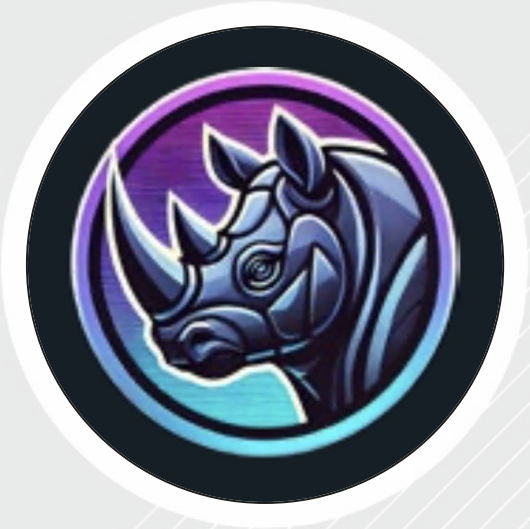
SPYWOLF

Security Audit Report



Audit prepared for

RhinoFi

Updated on

July 7, 2025

OVERVIEW

This goal of this report is to review the main aspects of the project to help investors make an informative decision during their research process.

You will find a a summarized review of the following key points:

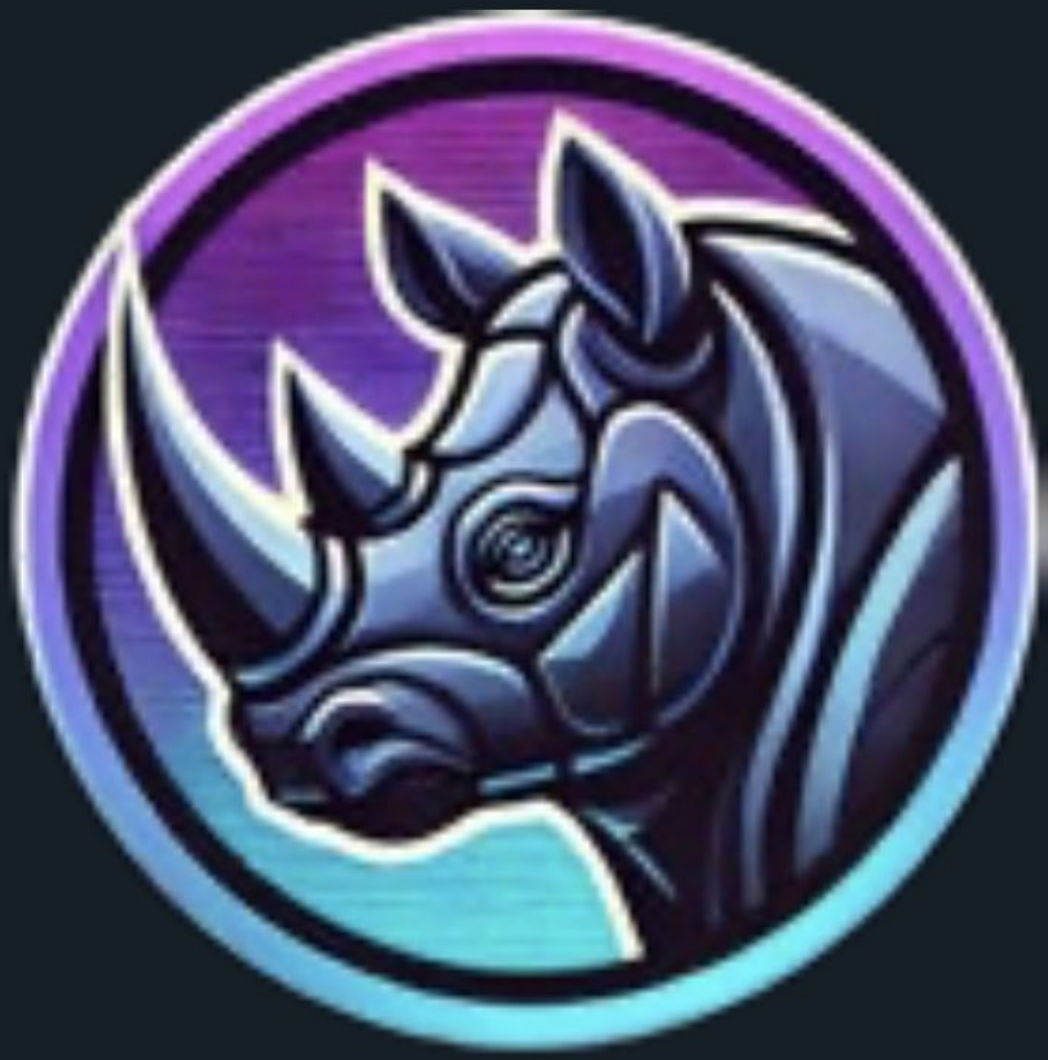
- Contract's source code
- Owners' wallets
- Tokenomics
- Team transparency and goals
- Website's age, code, security and UX
- Whitepaper and roadmap
- Social media & online presence

The results of this audit are purely based on the team’s evaluation and does not guarantee nor reflect the projects outcome and goal

- SPYWOLF Team -

TABLE OF CONTENTS

RHINOFI



PROJECT DESCRIPTION:

RhinoFi is a fee-based, hyper-deflationary reflection token protocol designed to reward long-term holders and liquidity providers while creating a sustainable ecosystem of value and utility. By leveraging an innovative dynamic sell tax structure, multi-asset reflections, and strategic liquidity incentives, RhinoFi ensures that users (Rhinos) benefit simply by holding RHINO tokens in their wallets.

Release Date: TBA

Launchpad: TBA

Category: DeFi/Dividend

CONTRACT(S) INFORMATION

RHINO.sol

- Manages user balances and transfers as the core ERC20 token of the protocol .
- Implements a dynamic tax system where fees decrease based on how long a user has held the tokens .
- Collects fees from transactions and uses a swapBack function to convert the collected RHINO into other assets.
- Distributes reflection rewards to token holders via its internal DividendDistributor contract .
- Acts as a trigger to initiate automated cycles in the RhinoCharging contract during user transfers.

RhinoCharging.sol

- Serves as the protocol's primary engine for yield-farming and value accrual.
- Accumulates various approved "reward tokens" that it receives.
- Periodically swaps these reward tokens into PLS via its triggerSwap function.
- Uses the acquired PLS to provide liquidity to a PulseX (DAI/WPLS) farm, earning additional rewards .
- Operates in "charging cycles" ; at the end of a cycle, it liquidates all assets to PLS and forwards the balance to the RhinoBuyAndBurn contract .

RhinoBuyAndBurn.sol

- Creates deflationary pressure on the RHINO token by using the PLS profits sent from the RhinoCharging contract.
- Allocates the received PLS for use in time-based intervals instead of all at once .
- Allows any external user to call the buyAndBurn function, which uses the allocated PLS to purchase RHINO tokens from the market.
- After purchasing RHINO, it burns half of the tokens and sends the other half back to the RhinoCharging contract .
- Rewards the user who triggers the buyback with a small incentive fee paid in PLS.

FOUND THREATS

~~High Risk~~ ✅ Mitigated

Broken Rewards Accounting for the Farming Contract

Description: This is a critical flaw in the protocols economic design. The RhinoCharging contract is intended to be a major holder of RHINO tokens and earn reflection rewards from various other assets. However the code lacks any mechanism to update the RhinoCharging contract's share balance in the DividendDistributor after its initial deployment. As it accumulates more RHINO through reflections or deposits, its reward-earning power remains stagnant at zero or its initial amount, preventing it from claiming the rewards it is owed.

Code Snippet (RhinoCharging.sol):

```
function depositRhinoFromBurn(uint256 amount) public {
    require(msg.sender == address(rhinoBuyAndBurn), "Only burning contract can deposit RHINO");
    IERC20(RHINO).safeTransferFrom(msg.sender, address(this), amount);
    rhinoChargingBalance += amount;
}
```

In this function, the contract receives RHINO tokens, and its internal rhinoChargingBalance is updated. However, there is no corresponding call to the DividendDistributor to update its shares, meaning its entitlement to reflection rewards does not increase.

Impact on Users: The protocol's primary value-generation engine is fundamentally broken. The inability to claim rewards means significantly less revenue is generated for farming and for the buy-and-burn mechanism. This directly harms users by weakening the deflationary pressure and price support that the system is designed to provide.

Developer's Comment:

"depositFromRhinoBurn would actually update the shares...the mere act of tokens being transferred to the rhino charging contract, updates it's shares, as seen in the _transferFrom function...To correct this, I have updated the launch() function to set the shares for the owner address as well as the RhinoCharging address."

Auditor's Verification:

The developer's analysis is correct. After a detailed re-examination of the code's execution paths, we can confirm that all implemented methods for the RhinoCharging contract to receive RHINO tokens (including deposits from the burn contract and claimed reflection rewards) correctly trigger the _transferFrom function. This function contains the necessary logic to update the recipient's dividend shares, ensuring the RhinoCharging contract's reward-earning power stays in sync with its balance. The developer's addition to the launch() function also successfully addresses the potential edge case for pre-launch deposits.

FOUND THREATS

~~High Risk~~ 🔵 Acknowledged

Value Leakage via MEV "Sandwich Attacks"

Description: The functions that perform automated tokens swaps are vulnerable to front running by MEV bots. The contract calculates its maximum tolerable slippage internally and then broadcasts the transaction to the network. Bots can see this predictable trade, manipulate the token price before the swap executes, and then sell immediately after, guaranteeing themselves a risk-free profit at the protocol's expense.

Code Snippet (RhinoBuyAndBurn.sol):

```
function _swapPlsForRhino(uint256 _plsAmount, uint256 _deadline) internal returns (uint256) {
    // ...
    uint[] memory expectedAmounts = IUniswapV2Router02(bestRouter).getAmountsOut(_plsAmount, path);
    uint256 expectedAmountOut = expectedAmounts[1];
    uint256 amountOutMin = (expectedAmountOut * (10_000 - slippageBps)) / 10_000;
    uint[] memory amounts = IUniswapV2Router02(bestRouter).swapExactETHForTokens{value: _plsAmount}(
        amountOutMin,
        //...
    );
}
```

The code first gets the expectedAmounts , calculates a predictable amountOutMin based on a stored slippageBps value , and only then executes the swap. This multi step process within a single function call is what makes it vulnerable.

Impact on Users: This vulnerability creates a constant "value leak" from the ecosystem. Every time the protocol performs a buy-and-burn, it will receive fewer RHINO tokens than it should have, with the value difference being captured by MEV bots. This dilutes the positive impact of every buyback, harming all token holders.

Developer's Comment:

"Acknowledged. On pulsechain there is no true way to protect slippage with V2 swaps that happen automatically...The MEV themselves can (and will) trigger this function...Rhino has a 6% buy tax and a sell tax up to 12%, which makes it incredibly hard for an outside MEV to be able to make a profit..."

Auditor's Verification:

The developer's reasoning is valid. High taxes do provide a strong deterrent against MEV, and this is a known challenge for automated swaps on V2 DEXs. The team has made a conscious business decision to accept this residual risk.

FOUND THREATS

~~⚠ High Risk~~ ✅ Mitigated

Potential System Freeze (Denial of Service)

Description: The _triggerSwap function in the RhinoCharging contract contains an "unbounded loop." It iterates thru the entire rewardTokens array within a single transaction to swap each token for PLS. The owner can add an unlimited number of tokens to this array.

Code Snippet (RhinoCharging.sol):

```
function _triggerSwap(uint256 deadline) internal inSwap {
    for (uint i = 0; i < rewardTokens.length; i++) {
        address currentToken = rewardTokens[i];
        // ... balance checks and swap logic ...
    }
    // ...
}
```

This for loop must process the entire rewardTokens array in one execution. As the array's length increases, the gas cost of the function grows linearly.

Impact on Users: If the list of reward tokens becomes too long, the gas required to execute this function will exceed the blockchain's block gas limit. This will cause the function to fail every time it is called, permanently freezing the protocol's ability to swap its reward tokens. This halts the entire automated farming and buy-back cycle, stopping the system from generating value for users.

Developer's Comment:

"I have removed the onlyOwner function addRewardToken(), this contract will be limited to the hard-set tokens in constructor deployment, to prevent any possible way the system could be DoS'd."

Auditor's Verification:

By removig the ability to add new reward tokens after deployment, the rewardTokens array now has a fixed, known maximum size. This makes the loop's gas cost predictable and eliminates the DoS vector.

FOUND THREATS

~~⚠ Medium Risk~~ 🔵 Acknowledged

Inconsistent Security Model and Re-entrancy Risk

Description: The ecosystem uses a mix of different methods to prevent re-entrancy attacks. Some functions use the industry standard ReentrancyGuard provided by OpenZeppelin, while another critical functions use custom-built boolean flags (e.g., inSwap, swapping). This inconsistency creates a more complex and fragile security posture, making the code harder to reason about and increasing the risk of a developer accidentally introducing a re-entrancy loophole in a future update.

Code Snippet (RhinoCharging.sol vs. RhinoBuyAndBurn.sol):
(This example shows two different approaches to the same problem.)

```
// In RhinoCharging.sol, a custom modifier is used
modifier inSwap() {
    swapping = true;
    _;
    swapping = false;
}

function _triggerSwap(uint256 deadline) internal inSwap {
    // ...
}
```

```
// In RhinoBuyAndBurn.sol, the standard ReentrancyGuard is used
function buyAndBurn(uint256 _deadline) external nonReentrant intervalUpdate {
    // ...
}
```

The protocol uses both the custom inSwap modifier and the standard nonReentrant guard for functions that perform swaps, leading to an inconsistent security design.

Impact on Users: While not a direct exploit now, this architectural weakness makes the protocol less secure over the long term. Future updates are more likely to contain errors that could lead to re-entrancy exploits, potentially allowing an attacker to drain funds or get the contract stuck.

Developer's Comment:

"Acknowledged. This is due to the unique features of tax tokens, to prevent the contract from reentry attacking itself, without causing a global revert all transfers "stuck" scenario...It is important for the long-term operability of the ecosystem for these modifiers and reentry guards to remain as they are..."

Auditor's Verification:

The developer has provided a specific architectural reason for this design choice, indicating it is deliberate. While it deviates from best practice, they are accepting the trade-off of increased complexity for their desired functionality.

FOUND THREATS

~~⚠️ Medium Risk~~ ✅ Finding Retracted

Transaction Failures from Hardcoded Deadlines

Description: Several functions that execute critical token swaps use the current block.timestamp as the transaction deadline. This leaves no room for network delay. If a transaction is submitted but not included in the very next block (e.g., due to network congestion), the deadline will have already past when it finally executes, causing the transaction to fail.

Code Snippet (RhinoCharging.sol):

```
function endChargingCycle() internal endingCycle {
    // ...
    uint256 daiBalance = IERC20(DAI).balanceOf(address(this));
    if(daiBalance > 0) {
        // ...
        try pulseXRouterV1.swapExactTokensForETH(daiBalance, amountOutMin, path, address(this), block.timestamp) {} catch {}
    }
    // ...
}
```

The final parameter in the swapExactTokensForETH call is the deadline, which is set to the current block.timestamp, providing no grace period for transaction mining.

Impact on Users: This makes the protocol unreliable, especially during periods of high network traffic. Users may have their transactions for ending a farming cycle fail repeatedly, forcing them to waste gas and delaying their access to the protocol's collected revenue.

Developer's Comment:

"I believe this finding to be untrue. The block.timestamp that will be passed into the internal swap functions, will always be the same timestamp for whatever block it eventually gets included in. It's not possible for this to revert for "EXPIRED" deadline."

Auditor's Verification:

The developer's technical reasoning is correct. For an internal contract-to-contract call that passes block.timestamp as the deadline parameter, the router's check (require(deadline >= block.timestamp)) will always pass. The finding as it relates to an "expired" revert is invalid.

FOUND THREATS

~~⚠ Low Risk~~ ✅ Mitigated

Theoretical Integer Overflow in Time Calculation

Description: In RhinoBuyAndBurn contract, the logic to calculate the number of missed processing intervals casts the result to a uint16 integer type. A uint16 can only hold a maximum value of 65,535. While this is sufficient for any normal operational scenario, it introduces a theoretical risk that the number could "overflow" if the function is not called for an extremely long period (several years).

Code Snippet (RhinoBuyAndBurn.sol):

```
function _intervalUpdatePlsForBurning() internal {
    // ...
    uint32 timeElapsed = uint32(block.timestamp - lastBurnIntervalStartTimestamp);
    if (timeElapsed < intervalTime) return;

    uint256 intervals_per_day = (24 hours) / intervalTime;
    uint16 missedIntervals = uint16(timeElapsed / intervalTime) - 1; // This line contains the cast to uint16
    //...
}
```

The missedIntervals variable is explicitly cast to uint16, which limits the number of intervals that can be correctly processed in a single update.

Impact on Users: The impact on users is negligible and purely theoretical. This issue would only manifest if the protocol was left completely inactive yet still holding funds for many years. In such an improbable scenario, the fund allocation logic for the buy-and-burn could fail or behave incorrectly.

Developer's Comment:

"I have updated all to uint256"

Auditor's Verification:

The code confirms this change has been made.

FOUND THREATS

Informational

Expanded Owner Privileges and Centralization

The new RhinoCharging and RhinoBuyAndBurn contracts add many new owner-only functions that control key economic parameters like cycle durations, farm thresholds, and slippage settings. This further centralizes control over the protocols operations, adding to the extensive privileges already present in the core RHINO contract.

Increased System Complexity and Attack Surface

The evolution from a single token contract to an interconnected three-contract ecosystem significantly increases the system's complexity. While this enables powerful automation, it also makes the codebase harder to reason about and expands the potential "attack surface" for future vulnerabilities.

Gas Optimization in Loops

In functions containing loops that read from a storage array, such as the for loop in RhinoCharging._triggerSwap, the array's length is accessed on each iteration. Caching the array length in a memory variable before the loop begins is a best practice that saves a small amount of gas and can improve code clarity.

Event Emission for Critical Failures

Functions that use a try/catch block to handle potential external call failures, like the swap attempts in RhinoCharging.sol, could benefit from emitting an event within the catch block. This would create an on-chain log of failed operations, which is valuable for off-chain monitoring and debugging.

Developer's Comment:

The developer acknowledged the centralization and complexity as necessary, implemented the gas optimizations, and confirmed the lack of events was a deliberate choice to save gas.

Auditor's Verification:

The developer's feedback is noted. The gas optimization of caching the array length in _triggerSwap has been confirmed in the code.

AUDIT METHODOLOGY

The audit was conducted in the following phases:

1. **Architectural Review & Scoping**
   - The initial phase involved a thorough review of all provided documentation, including the whitepaper, developer comments, and the three core smart contracts: RHINO.sol, RhinoCharging.sol, and RhinoBuyAndBurn.sol. The primary goal was to understand the intended business logic and the complex interactions within the automated farm-to-burn cycle.
2. **Automated Static Analysis**
   - We utilized industry-standard static analysis tools, such as Slither and Mythril, to perform an initial scan of the codebase. This process checks for common, known vulnerability patterns as defined by the Smart Contract Weakness Classification (SWC) registry. This phase serves as a baseline but is insufficient for detecting the complex logic flaws this audit later uncovered.
3. **Manual Code Review**
   - This was the most critical phase of the audit, consisting of a meticulous line-by-line examination of the contracts. Our focus was on identifying flaws that automated tools typically miss, including:
     - **Economic Logic:** Verifying the correctness of the rewards accounting, fee distribution, and the farm-to-burn value flow.
     - **Security Architecture:** Analyzing how the system manages state and prevents race conditions, with a focus on the mix of custom locks and standard ReentrancyGuard patterns.
     - **External Interactions:** Assessing the safety and reliability of interactions with external protocols like Uniswap V2.
4. **Simulated Exploit & Functionality Testing**
   - To validate vulnerabilities suspected during the manual review, we conducted targeted tests within a Hardhat environment forking the PulseChain mainnet. This included:
     - **Simulating an MEV sandwich attack** to prove the value leakage from the automated swap functions .
     - **Confirming the DoS vulnerability** by creating a test to populate the rewardTokens array and observing the consistent gas failure of the triggerSwap function .
     - **Validating the rewards accounting flaw** by checking the RhinoCharging contract's earnings, which remained at zero until its shares were manually updated, proving the automated mechanism was broken .
5. **Mitigation Review & Reporting**
   - The final phase involved reviewing the developer's feedback and code changes against our initial findings. The final report was then compiled, with each vulnerability's status updated to reflect whether it was Mitigated or Acknowledged as an accepted risk.

CONCLUSION

This re-audit of the RhinoFi ecosystem has been a highly collaborative and productive process. We commend the development team for their diligent and responsive approach, which has resulted in a significant improvement to the protocol's overall security posture. Key vulnerabilities, including a critical Denial of Service vector, have been successfully mitigated, demonstrating a strong commitment to building a secure platform. The addition of new safety features like emergencyMode further shows a thoughtful consideration for user security in worst-case scenarios.

While the protocol is now substantially more robust, our analysis confirms that the development team has successfully mitigated the critical vulnerabilities identified, including the core economic logic flaw related to rewards accounting and the risk of a system freeze. Additionally, the team has made a conscious and well-documented decision to accept the risks associated with MEV Value Leakage and their Inconsistent Security Model as necessary trade-offs for their desired functionality.

SPYWOLF

CRYPTO SECURITY

ABOUT US

We are a growing crypto security agency offering audits, KYCs and consulting services for some of the top names in the crypto industry.

- **OVER 700 SUCCESSFUL CLIENTS**
- **MORE THAN 1000 SCAMS EXPOSED**
- **MILLIONS SAVED IN POTENTIAL FRAUD**
- **PARTNERSHIPS WITH TOP LAUNCHPADS, INFLUENCERS AND CRYPTO PROJECTS**
- **CONSTANTLY BUILDING TOOLS TO HELP INVESTORS DO BETTER RESEARCH**

Audits | KYCs | dApps
Contract Development

To hire us, reach out to
contact@spywolf.co or
t.me/joe_SpyWolf

FIND US ONLINE

SPYWOLF.CO

@SPYWOLFNETWORK

@SPYWOLFNETWORK

Disclaimer

This report shows findings based on our limited project analysis, following good industry practice from the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, overall social media and website presence and team transparency details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report.
While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER:
By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis, and does not constitute investment advice.
No one shall have any right to rely on the report or its contents, and SpyWolf and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives) (SpyWolf) owe no duty of care towards you or any other person, nor does SpyWolf make any warranty or representation to any person on the accuracy or completeness of the report.
The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and SpyWolf hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, SpyWolf hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against SpyWolf, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts, website, social media and team.
No applications were reviewed for security. No product code has been reviewed.