QuillAudits

AUDIT REPORT

June 2025

For

pyro

Table of Content

Executive Summary

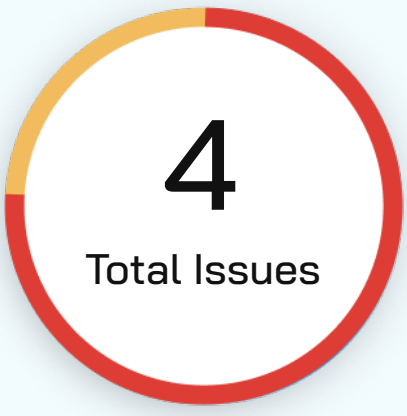
Project Name	Pyro fun
Protocol Type	Token Launcher
Project URL	https://pyro.fun
Overview	Pyro Fun is token launcher platform built on Solana that combines meme coin creation with DeFi mechanics. The platform allows users to launch new SPL tokens with customizable parameters (name, symbol, metadata URI) and automatically sets up a bonding curve-based AMM for trading. Each launched token gets its own liquidity pool where users can buy/sell tokens at prices determined by the constant product formula (x * y = k). The system includes a unique burn-to-earn staking mechanism - users can burn their original tokens to receive “burned tokens” which can then be staked in farms to earn SOL rewards. Trading fees are split between the development team and staking rewards, creating sustainable tokenomics. The platform supports standard DeFi operations like buying, selling, staking, unstaking, and claiming rewards, with comprehensive event logging for frontend integration.
Audit Scope	The scope of this Audit was to analyze the Pyro fun Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/tkzo/pyro-contracts
Contracts in Scope	src/*
Branch	Main
Commit Hash	ef5d3f04ad365366555e746c7a93856517dee896
Language	Rust
Blockchain	Solana
Method	Manual Analysis, Functional Testing, Automated Testing

Review 1	16th June 2025 - 28th June 2025
Updated Code Received	30th June 2025
Review 2	30th June 2025 - 3rd June 2025
Fixed In	4960098fce26a3e4223db2af488379d4967789a3

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0 (0%)
High	3 (25%)
Medium	0 (0%)
Low	1 (25%)
Informational	0 (0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	**3**	0	**1**	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Double Accounting of rewards	High	Resolved
2	Missing mut Constraint in modify_launcher	High	Resolved
3	Broken Burn CPI Signer	High	Resolved
4	Unnecessary data allocation for liquidity vault	Low	Resolved

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls

- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw
- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of Solana Programs, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

High Severity Issues

Double Accounting of rewards

Resolved

Path

/pyro-contracts-main/programs/pyro/src/helpers.rs

Function

earned()

Description

The `earned()` function in the Pyro contract incorrectly calculates staking rewards by failing to use the user's checkpoint (`user_reward_per_token`), allowing users to repeatedly claim the same rewards multiple times, leading to excessive reward distribution and potential drainage of the reward vault.

The current implementation of the `earned()` function does not account for rewards that have already been included in previous calculations, resulting in double-counting of rewards with each claim.

Current Implementation

The function uses the total cumulative `reward_per_token` value but doesn't subtract the user's previous checkpoint (`user_reward_per_token`). This means users get rewards for the entire history of the farm with each claim.

Example Scenario

Example 1: First Stake (Bug Not Yet Apparent)
- User stakes 100 tokens at t=0
- Reward rate: 10 rewards/second
- At t=500 (first claim):
 - `reward_per_token = 50 * scale` (cumulative)
 - `user_reward_per_token = 0` (initial value)
 - Current calculation: `(100 * 50 * scale) / scale + 0 = 5000`
 - User claims 5000 rewards ✓ (correct first time)

Example 2: Second Claim (Bug Manifests)**
 - At t=1000 (second claim):
 - `reward_per_token = 100 * scale` (cumulative total)
 - `user_reward_per_token = 50 * scale` (from last update)
 - Current (Wrong) Calculation: `(100 * 100 * scale) / scale + 0 = 10000`
- Should Be: `(100 * (100-50) * scale) / scale + 0 = 5000`
 - Result: User gets 10000 instead of 5000 (double rewards!)

Example 3: Third Claim (Bug Compounds)
- At t=1500 (third claim):
 - `reward_per_token = 150 * scale` (cumulative)
 - `user_reward_per_token = 100 * scale` (from last update)
 - Current (Wrong) Calculation: `(100 * 150 * scale) / scale + 0 = 15000`
 - Should Be: `(100 * (150-100) * scale) / scale + 0 = 5000`
 - Result: User gets 15000 instead of 5000 (triple rewards!)

Impact

- Protocol Insolvency: Users can claim far more rewards than intended, potentially draining the entire rewards vault.
- Exponential Exploitation: The more frequent the claims, the worse the exploitation, allowing attackers to maximize reward extraction.
- Unfair Distribution: Early or frequent claimers receive disproportionately large rewards compared to others.

Remediation

Modify the `earned()` function to properly subtract the user's checkpoint from the current reward per token.

```rust
fn earned(farm: &Farm, stake_info: Option<&mut StakeInfo>) -> Result<u128> {
    let stake = stake_info.unwrap();
    let balance = stake.amount;
    let reward_per_token = reward_per_token(farm)
        .unwrap()
        .checked_sub(stake.user_reward_per_token)
        .unwrap();
    let rewards = stake.rewards_earned;
    return Ok(balance
        .checked_mul(reward_per_token)
        .unwrap()
        .checked_div(farm.scale)
        .unwrap()
        .checked_add(rewards)
        .unwrap());
}
```

Missing mut Constraint in modify_launcher

Resolved

Path

src/lib.rs

Function

modify_launcher(...)

Description

In the ModifyLauncher instruction, the launcher account lacks the required mut constraint despite being mutated in the function body. The constraint validation fails at runtime when attempting to modify the account state, causing 100% failure rate for all modify_launcher calls.

Current Implementation

The ModifyLauncher context declares:

```
#[account(
    seeds = [b"launcher", b"v1".as_ref()],
    bump,
)]
pub launcher: Account<'info, TokenLauncher>,  // ← missing mut constraint
```

but the function attempts to mutate launcher fields

Attack Path

1.Deploy contract with missing mut constraint
2.Call modify_launcher with any parameters
3.Observe runtime failure due to constraint violation
4.Functionality becomes completely unusable

Remediation

Add mut constraint to launcher account:

```
#[account(
        mut,
        seeds = [b"launcher", b"v1".as_ref()],
        bump,
    )]
    pub launcher: Account<'info, TokenLauncher>
```

Broken Burn CPI Signer

Resolved

Path

programs/pyro/src/instructions/burn.rs

Function

burn_tokens()

Description

To burn the user's SPL tokens, the code builds a CPI with a signer seeds array for the launcher PDA, but passes the user (burner) as the authority account. The PDA seeds do not match the user, so the runtime signature check fails:

```
let seeds = &[b"launcher", b"v1".as_ref(), &[ctx.bumps.launcher]];
let signer = &[&seeds[..]];
let burn_accounts = BurnChecked {
    mint: ctx.accounts.token_mint.to_account_info(),
    from: ctx.accounts.burner_token_account.to_account_info(),
    authority: ctx.accounts.burner.to_account_info(),  // user, not PDA
};
let burn_ctx = CpiContext::new_with_signer(
    ctx.accounts.token_program.to_account_info(),
    burn_accounts,
    signer,  // signer seeds for PDA
);
burn_checked(burn_ctx, amount, 9)?; // ← will fail
```

Code Snippet

```
let burn_ctx = CpiContext::new_with_signer(
    ctx.accounts.token_program.to_account_info(),
    BurnChecked {
        authority: ctx.accounts.burner.to_account_info(),
        // …
    },
    signer,  // seeds for launcher PDA, not burner
);
burn_checked(burn_ctx, amount, 9)?;
```

Attack Path

-Mint some tokens to user's burner_token_account.
-Call burn_tokens(amount).
-Observe immediate CPI failure — transaction reverts with "signature verification" error.

Remediation

Remove the PDA signer seeds so that the user's own signature is used:

```
et burn_ctx = CpiContext::new(
    ctx.accounts.token_program.to_account_info(),
    BurnChecked {
      authority: ctx.accounts.burner.to_account_info(),
      mint: ctx.accounts.token_mint.to_account_info(),
      from: ctx.accounts.burner_token_account.to_account_info(),
    },
);
burn_checked(burn_ctx, amount, 9)?;
```

If you do need a PDA as authority, pass the PDA's account and seeds consistently.

Low Severity Issues

Unnecessary data allocation for liquidity vault

Resolved

Path

/pyro-contracts-main/programs/pyro/src/launch.rs

Function

launch_token

Description

The `launch_token` function over-allocates rent for the liquidity vault by calculating rent exemption for 8 bytes of data when the account doesn't need any data storage. Users unnecessarily pay this amount of sol each time they launch a token.

In the `launch_token` function, the code pre-funds the liquidity vault with enough SOL to make it rent-exempt, but it incorrectly assumes the account needs 8 bytes of data:

```rust
let rent_exemption = Rent::get()?.minimum_balance(8);
```

However, the liquidity vault is declared as an `UncheckedAccount` that simply holds SOL and doesn't store any data. This is evident from the comment and declaration:

```rust
/// CHECK: This is just a SOL receiver account
#[account(
    mut,
    seeds = [b"liquidity_vault", token_mint.key().as_ref()],
    bump
)]
pub liquidity_vault: UncheckedAccount<'info>,
```

For comparison, the rewards vault (which serves the same purpose) correctly uses 0 bytes:

```rust
let rent_exemption_rewards = Rent::get()?.minimum_balance(0);
```

Impact

This issue results in:
- Inefficient use of funds – a small amount of extra SOL is locked in each liquidity vault unnecessarily
- Inconsistency between the rewards vault and liquidity vault funding mechanisms
- No security vulnerability, but represents a minor problem where users pay a little extra than necessary.

Impact

Change the rent calculation for the liquidity vault to use 0 bytes instead of 8 bytes:

```rust
let rent_exemption = Rent::get()?.minimum_balance(0);
```

This change will make the code more consistent and efficient by only allocating the minimum necessary SOL for rent exemption.

Functional Tests

Some of the tests performed are mentioned below:

Token Launching Tests

- Test successful token launch with valid parameters
- Test token launch with different initial supply and liquidity values
- Test metadata creation with proper name, symbol, and URI
- Test vault accounts are initialized correctly
- Test all PDAs are derived with the correct seeds
- Test rent exemption for created accounts

Bonding Curve Tests

- Test buy increases liquidity_reserve and decreases token_reserve
- Test sell decreases liquidity_reserve and increases token_reserve
- Test invariant (liquidity_reserve * token_reserve = k) holds after operations
- Test buying with min_amount_out parameter works (rejects if too low)
- Test selling with min_amount_out parameter works (rejects if too low)
- Test fee distribution between dev and farm
- Test slippage with different transaction sizes

Farm Reward Tests

- Test reward calculation for different time periods
- Test reward accumulation works correctly over time
- Test rewards distribution with multiple users staking
- Test reward_rate updates after buys/sells
- Test claiming reduces rewards vault balance
- Test reward accounting with the buffer mechanism

Stake/Unstake Tests

- ✔ Test staking increases farm.total_staked
- ✔ Test unstaking decreases farm.total_staked
- ✔ Test stake_info is updated correctly
- ✔ Test multiple stake/unstake operations for the same user
- ✔ Test rewards are updated before stake/unstake operations

Burn Tests

- ✔ Test burning tokens correctly mints equivalent burned tokens
- ✔ Test burned tokens can be staked in the farm
- ✔ Test burn operation requires proper approvals
- ✔ Test burning tokens affects total supply

Admin Tests

- ✔ Test modifying launcher parameters updates correctly
- ✔ Test parameter changes affect future operations appropriately
- ✔ Test unauthorized users cannot modify launcher parameters
- ✔ Test fee structure changes work as expected

Threat Model

```
## TokenLauncher

### Function: initialize_launcher

- Unchecked parameter values (fee_share_maximum, etc.)
- Centralized admin/authority risk
- Incorrect rent exemption calculation for vaults

### Function: modify_launcher

- Centralized admin/authority risk
- Parameter manipulation (fees, durations)
- No timelock or multi-sig for critical changes

## Farm

### Function: announce_rewards

- Over-accounting of rewards if not properly synchronized with vault funding
- Integer overflow/underflow in reward math
- Reward rate manipulation by timing of calls

### Function: update_rewards / earned

- Double counting if user checkpoint not used (see H-01)
- Integer overflow/underflow
- Division by zero if total_staked is zero
- Precision loss due to integer division

## BondingCurve

### Function: buy_tokens

- Price manipulation via large trades
- Slippage exploitation (min_amount_out)
- Fee calculation errors
- Invariant violation if math is incorrect
- Integer overflow in curve math

### Function: sell_tokens

- Invariant violation (x*y=k)
- Flash loan attacks to manipulate price
- Drain attacks via coordinated sells
- Fee avoidance

## StakingSystem

### Function: stake_tokens / unstake_tokens

- Reward sniping (stake/unstake around reward events)
- Account initialization race conditions
- Free riding (stake just before, unstake just after rewards)
- Unstaking without staking (logic flaws)
```

```
### Function: claim_rewards

- Double claiming (see H-01)
- Reward vault insolvency (over-accounting)
- Integer overflow in reward calculation
- No check for sufficient vault balance before transfer

## TokenBurner

### Function: burn_tokens

- 1:1 conversion regardless of price (unfair staking power)
- Burning without permission (authorization issues)
- Minting failure after burn

## General

- Unauthorized admin functions
- Rent exemption draining
- Account validation bypass
- Denial of service (resource exhaustion)
- Arithmetic overflow/underflow
- Cross-program invocation attacks
- PDA address collision
```

Automated Tests

No major issues were found. Some false-positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of Pyro fun. We performed our audit according to the procedure described above.

issues of High and Low severity were found. Pyro fun team resolved all the issues

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

June 2025

For

pyro

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com