contact@madshield.xyz 19 Dec, 2024

Carrot Program Audit Report

– Mad Shield

David P – *david@fomo3d.app david@madshield.xyz*
BlueWolf – *wolf@madshield.xyz*

Table of Contents

1. Introduction

This audit focuses on the Carrot protocol, a Solana-based protocol that creates investment strategies by integrating with other protocols like Mango Markets, Drift, Klend and Marginfi as yield sources.

The audit was conducted between 25th September 2024 and 18th December 2024. During this period, the Carrot program was thoroughly analyzed with particular attention to cross-program invocations, vault management, and integration security with external protocols.

This report outlines the audit process, describes the methodology used, and certifies the program as secure to the best of our knowledge, while also highlighting areas for potential optimization and enhancement. The audit examines the incentive structures created by the protocol's strategy system and its interaction with external yield sources.

2. Findings & Recommendations

Our severity classification system adheres to the criteria outlined here.

Severity Level	Exploitability	Potential Impact	Examples
Critical	Low to moderate difficulty, 3rd-party attacker	Irreparable financial harm	Direct theft of funds, permanent freezing of tokens/NFTs
High	High difficulty, external attacker or specific user interactions	Recoverable financial harm	Temporary freezing of assets
Medium	Unexpected behavior, potential for misuse	Limited to no financial harm, non-critical disruption	Escalation of non-sensitive privilege, program malfunctions, inefficient execution
Low	Implementation varia(nce, uncommon scenarios	Zero financial implications, minor inconvenience	Program crashes in rare situations, parameter adjustments by authorized entities
Informational	N/A	Recommendations for improvement	Design enhancements, best practices, usability suggestions

In the following, we enumerate some of the findings and issues we discovered and explain their implications and corresponding resolutions.

Finding	Description	Severity Level
CARROT_01	Incorrect Space Allocation for DriftInsuranceFund Strategy Initialization	*Informational*
CARROT_02	Single Oracle Implementation	*Informational*

No, Critical, High, Medium or Low severities were identified.

2.1 Informational

[CARROT_01] - Incorrect Space Allocation for DriftInsuranceFund Strategy Initialization

Description

An issue has been identified in the initialization process of the `DriftInsuranceFund` strategy within the Carrot protocol. The `DriftInsuranceFundStrategyInit` struct, responsible for initializing the strategy account, is allocating an incorrect amount of space.

The cause of this issue lies in the space parameter of the `#[account]` attribute macro for the strategy field. Currently, it's using `StrategyTypeSelection::DriftSupply` to determine the space allocation, which is inconsistent with the intended `DriftInsuranceFund` strategy type.
This mismatch leads to several potential impacts:

Potential Transaction Cost Increase: Solana's rent-burn mechanism, which is based on the total state size of accounts touched in a transaction, may lead to slightly higher costs when interacting with this oversized account.

Consistency and Maintenance Issues: The mismatch between the strategy type and its allocated space could lead to confusion or bugs in future development and maintenance efforts.

Masked Flexibility Concerns: If the DriftInsuranceFund strategy needs to store more data in the future, it might inadvertently work due to the extra space, potentially masking underlying design issues.

Recommendation

To resolve this issue, the space allocation should be corrected to match the intended strategy type. The fix is straightforward and involves changing the `StrategyTypeSelection` enum value used in the space parameter to `DriftInsuranceFund`.

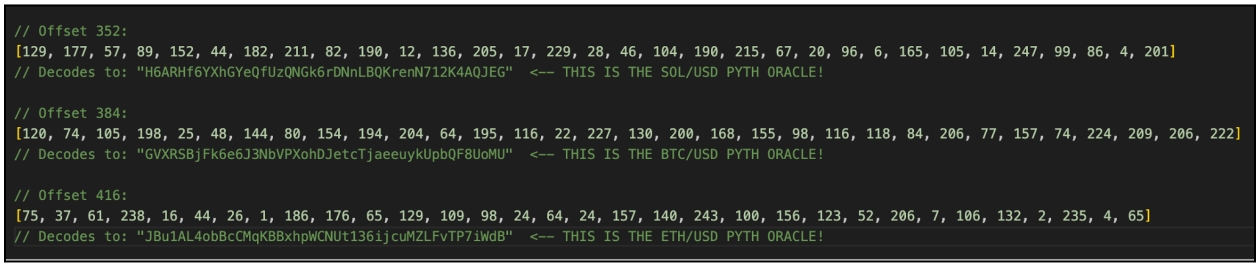
This change ensures that the correct amount of space is allocated, aligning the account size with its intended purpose and optimizing resource usage.

[CARROT_02] - Single Oracle Implementation

Description

Carrot Protocol currently implements Pyth as its sole oracle provider for price feeds, handling all asset valuations and calculations within the protocol. While Pyth is a reliable and well-established oracle service, the dependency on a single price feed source presents a security consideration that warrants attention.
Reliance on a single oracle: The reliance on a single oracle provider introduces several potential vulnerabilities to the protocol's operations. A single point of failure exists where any technical issues, network delays, or data inaccuracies in Pyth's service could directly impact the protocol's ability to accurately value assets and execute operations.

Pyth oracle check : We checked the price feed addresses stored in the account data to verify if the right ones were used.



```
// Offset 352:
[129, 177, 57, 89, 152, 44, 182, 211, 82, 190, 12, 136, 205, 17, 229, 28, 46, 104, 190, 215, 67, 20, 96, 6, 165, 105, 14, 247, 99, 86, 4, 201]
// Decodes to: "H6ARHf6YXhGYeQfUzQNGk6rDNnLBQKrenN712K4AQJEG"  <-- THIS IS THE SOL/USD PYTH ORACLE!

// Offset 384:
[120, 74, 105, 198, 25, 48, 144, 80, 154, 194, 204, 64, 195, 116, 22, 227, 130, 200, 168, 155, 98, 116, 118, 84, 206, 77, 157, 74, 224, 209, 206, 222]
// Decodes to: "GVXRSBjFk6e6J3NbVPXohDJetcTjaeeuykUpbQF8UoMU"  <-- THIS IS THE BTC/USD PYTH ORACLE!

// Offset 416:
[75, 37, 61, 238, 16, 44, 26, 1, 186, 176, 65, 129, 109, 98, 24, 64, 24, 157, 140, 243, 100, 156, 123, 52, 206, 7, 106, 132, 2, 235, 4, 65]
// Decodes to: "JBu1AL4obBcCMqKBBxhpWCNUt136ijcuMZLFvTP7iWdB"  <-- THIS IS THE ETH/USD PYTH ORACLE!
```

Recommendation
Implementing a multi-oracle approach would significantly enhance the protocol's security and reliability. By integrating additional oracle providers alongside Pyth, the protocol could implement price cross-validation and establish fallback mechanisms for continuous operation. This enhancement would provide protection against potential manipulation attempts and ensure more accurate price discovery through consensus mechanisms.

3. Protocol Overview

Carrot is a decentralized finance protocol built on Solana that introduces a professional asset management system through its vault architecture. At its core, the protocol operates through a vault system that acts as a managed fund, allowing users to deposit their assets while professional managers deploy various investment strategies.

The vault system serves as the foundation of the protocol, managing user deposits and executing investment strategies. When users deposit their assets into a Carrot vault, they receive share tokens that represent their proportional ownership of the vault's total assets. These share tokens are crucial as they track user ownership and ensure fair distribution of returns generated by the vault's investment activities.

One of Carrot's key features is its multi-asset support system. The protocol can handle various types of tokens, each tracked through Pyth Network price feeds that provide real-time valuation data. This price oracle integration is essential for accurate share calculations and informed strategy decisions, ensuring that user deposits and withdrawals are processed at fair market values.

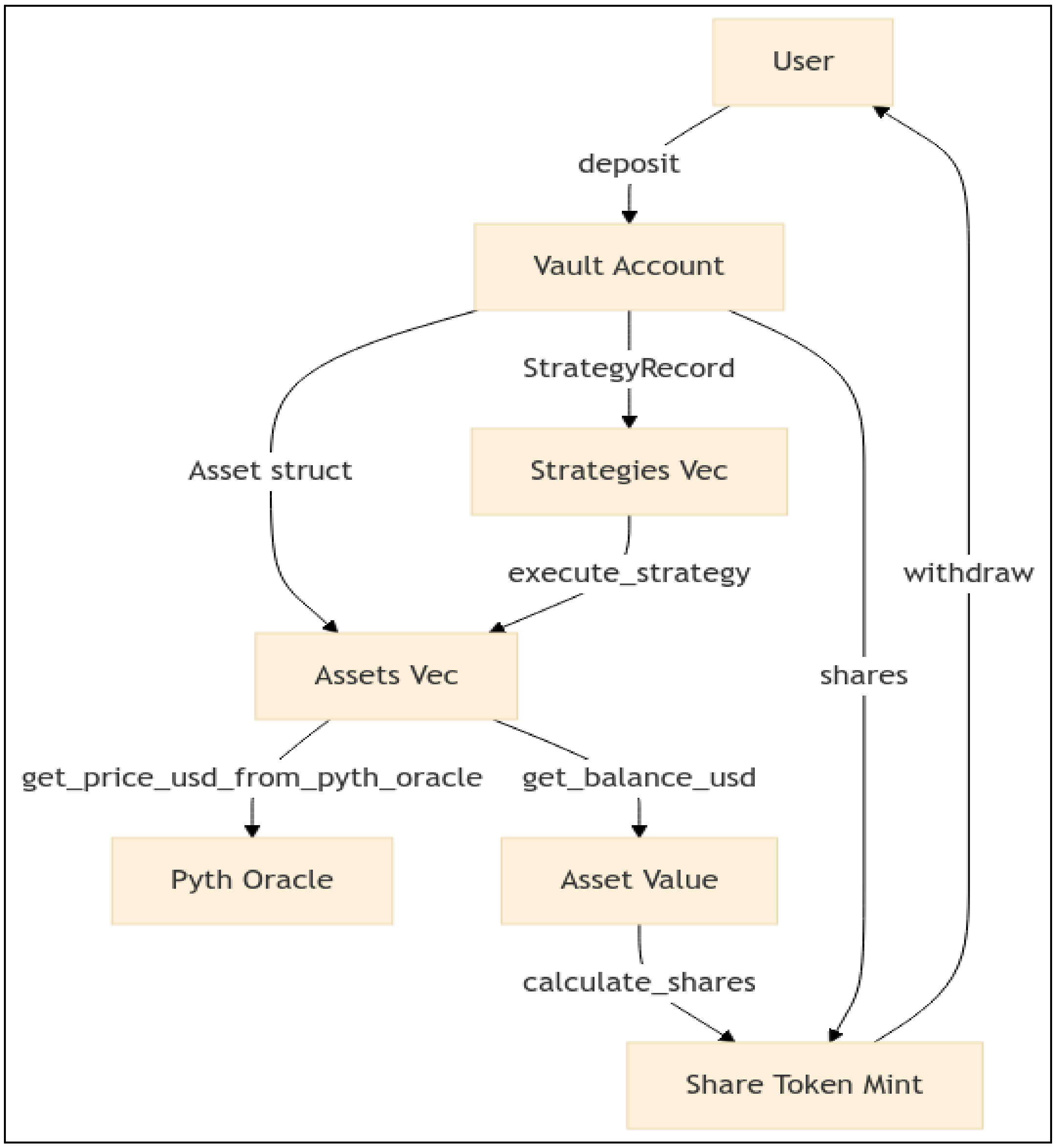
The strategy implementation aspect of Carrot is handled by professional managers who deploy vault assets across different investment opportunities. These strategies are designed to generate returns for users while maintaining appropriate risk management through diversification. The vault's authority controls these operations, ensuring that strategies are executed according to predetermined parameters and risk assessments.

The protocol's share token system is particularly noteworthy as it provides a seamless mechanism for managing user participation. When users deposit assets, they receive share tokens proportional to their contribution. These tokens can later be redeemed to withdraw assets, with the withdrawal amount reflecting their proportion of the vault's total value, including any returns generated through the deployed strategies.

3.2. Program Charts

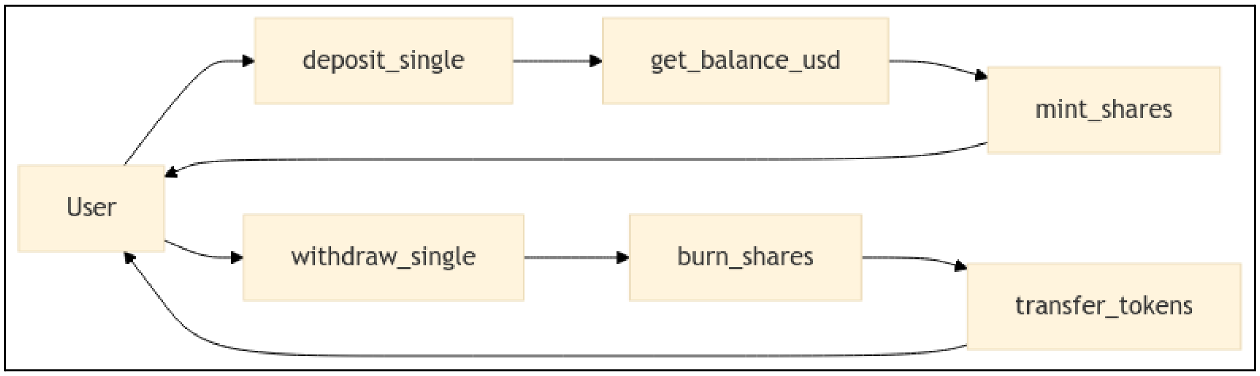
Vault Architecture Flow

The vault serves as the central component of Carrot, managing user deposits and strategy execution. It maintains a collection of supported assets and their corresponding strategies. When users interact with the vault, it handles share token minting/burning to track ownership proportions, while the authority manages strategy deployment and execution.



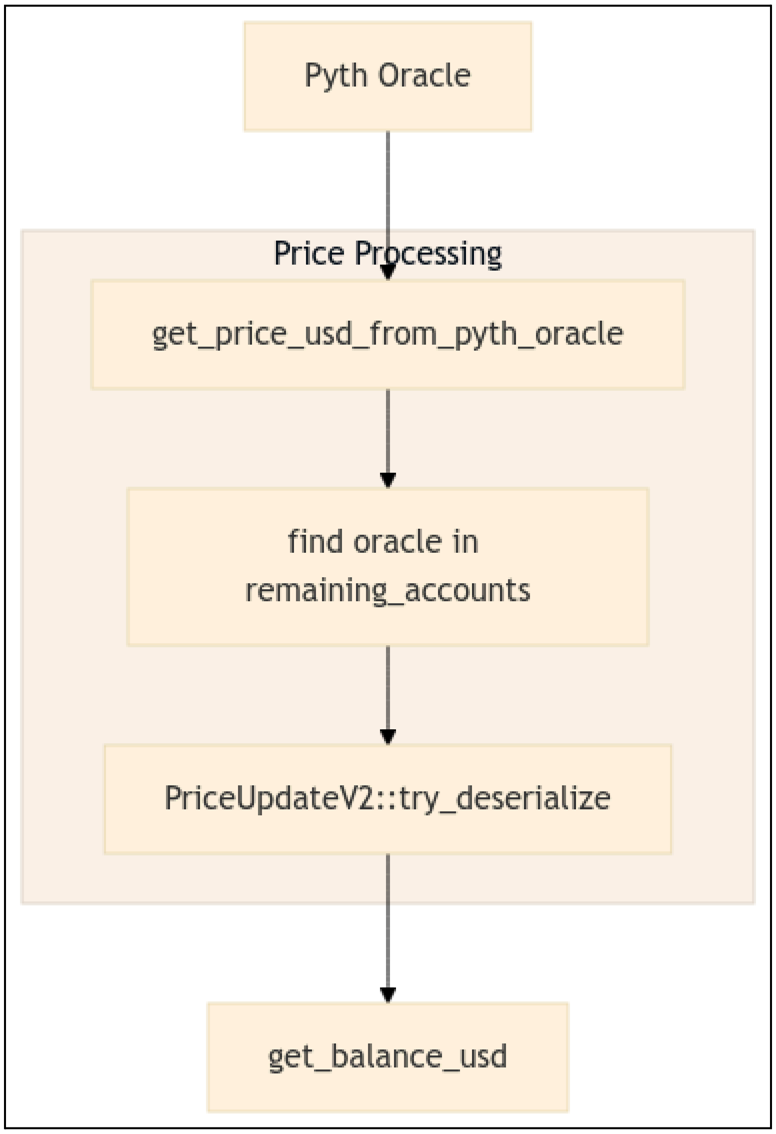
Asset Management Flow

Asset management in Carrot follows a precise flow where user deposits are processed through value calculations using Pyth oracle price feeds. The protocol calculates share tokens based on the deposit value relative to the total vault value. For withdrawals, the process reverses: share tokens are burned, and assets are returned based on the user's proportional ownership of the vault.



Oracle Integration Flow

Carrot integrates with Pyth oracles for real-time price feeds, essential for accurate asset valuation. The `get_price_usd_from_pyth_oracle` function fetches price data from Pyth oracle accounts, validates the data through account verification, and processes it for asset value calculations. This price data is crucial for share token calculations and strategy decisions.

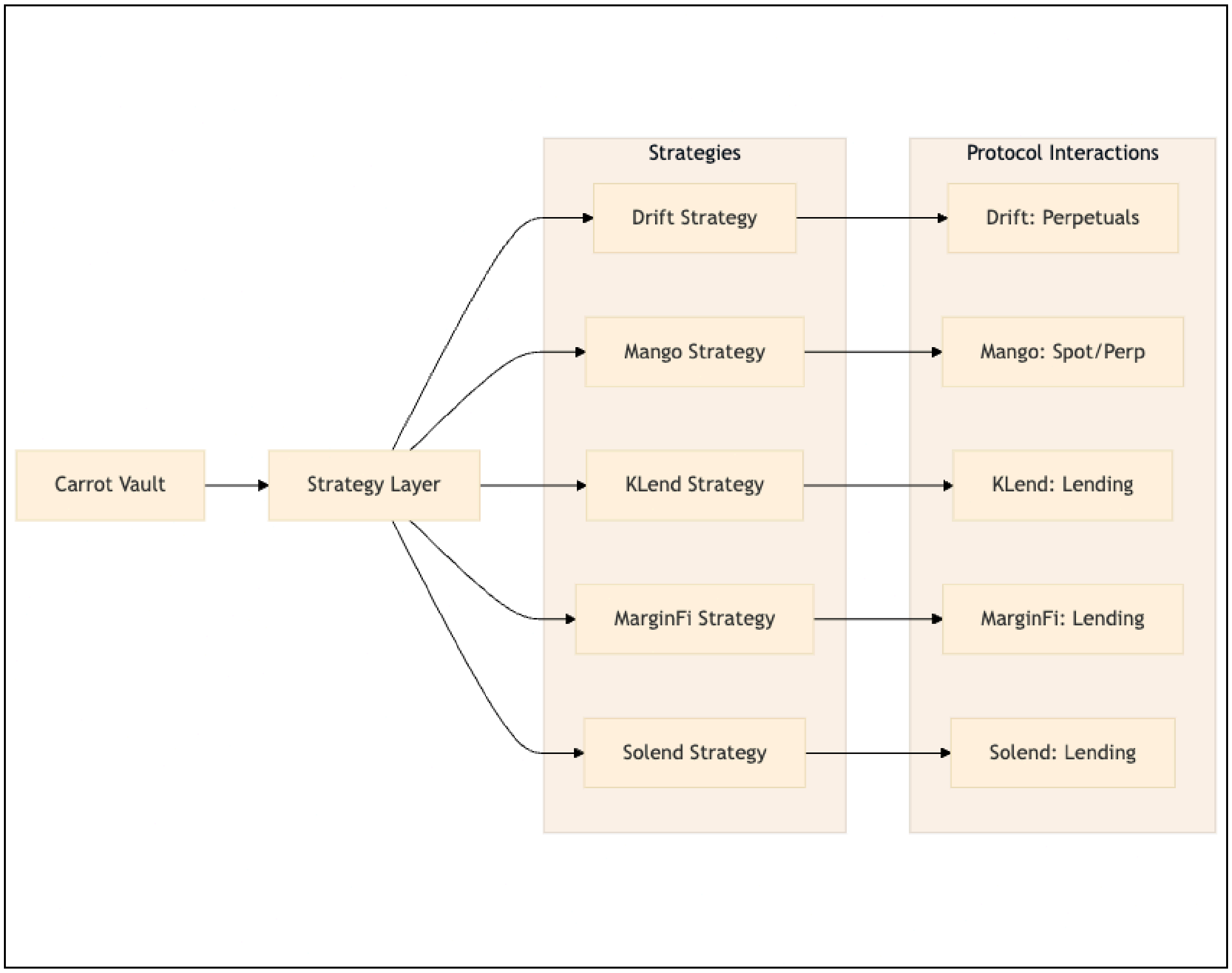

Pyth Oracle
Price Processing
get_price_usd_from_pyth_oracle
find oracle in remaining_accounts
PriceUpdateV2::try_deserialize
get_balance_usd

DeFi Integrations

Carrot interfaces with multiple DeFi protocols to execute various strategies:

- **Drift & Mango:** For perpetual futures and spot trading
- **KLend, MarginFi, & Solend:** For lending operations and yield generation

Each protocol integration enables specific strategy types, allowing the vault to diversify its operations across different DeFi opportunities while managing risk through strategy allocation.



4. Methodology

Given that the program is a vault protocol managing multiple assets, we employed a comprehensive and systematic methodology to test the protocol's constraints and behavior. Our primary approach involved analyzing the Pyth oracle integration and asset management system to validate the protocol's security and reliability.

This methodology allowed us to observe how the protocol would operate in real-world conditions, especially focusing on the oracle price feed implementation and asset value calculations. This security-focused approach was essential for evaluating the safety and reliability of the protocol's core components.

We conducted detailed code analysis of oracle interactions and asset management functions. This included reviewing the implementation of price feeds, asset value calculations, oracle account validation, and the security of vault operations across different scenarios.

The code review focused specifically on the oracle integration implementation, examining how price feeds are validated and used within the protocol. This helped in identifying potential security considerations that should be addressed, particularly around oracle account validation and multi-oracle implementation possibilities.

Throughout the analysis, we documented security observations and potential improvements to enhance the protocol's reliability and safety, ensuring robust operation in production environments.

5. Scope and Objectives

The primary objectives of the audit are defined as:

- Minimizing the possible presence of any critical vulnerabilities in the program. This would include detailed examination of the code and edge case scrutinization to find as many vulnerabilities.
- 2-way communication during the audit process. This included for Mad Shield to reach a perfect understanding of the design of the system and the goals of the team.
- Provide clear and thorough explanations of all vulnerabilities discovered during the process with potential suggestions and recommendations for fixes and code improvements.
- Clear attention to the documentation of the vulnerabilities with an eventual publication of a comprehensive audit report to the public audience for all stakeholders to understand the security status of the programs.

The Vault has delivered the program to Mad Shield at the following Github repositories.

Repository URL	https://github.com/DeFi-Carrot/protocol/
Commit (start of audit)	e0e7cdf57ec177187d820e02fa463a1a8a1aa125
Commit (end of audit)	10e4c86caf42e18b47f5222628c1872f1a41f2a4

Tab1. Audit Marks of Carrot

6. Conclusion

Mad Shield conducted an extensive audit of Carrot, utilizing a hands-on methodology that prioritizes a detailed, immersive review of the program. Our team's approach is rooted in active collaboration, working closely with each unique project to identify potential security risks and mitigate vulnerabilities effectively.

Mad Shield's dedication to advancing auditing techniques is clear throughout our process. We consistently apply innovative strategies, allowing us to analyze the code at a granular level, simulate real-world scenarios, and uncover potential risks that traditional audits might miss.

No critical vulnerabilities were identified during the Carrot audit, and all findings were promptly communicated to the development team. Our recommendations focus on strengthening the codebase to enhance long-term security and resilience. Mad Shield remains committed to setting new standards in smart contract auditing.