ANASTASIA LABS

Security Audit Report

Business Confidential

Date: April 17, 2024
Project: Genius Yield DEX
Version 1.0

Contents

Confidentiality statement

Scope and Disclaimer

The scope of this audit is limited to the on-chain code specified in the files audited section. The code responsible for the off-chain transaction construction and the back-end infrastructure was not reviewed in this audit.

A code review is considered a snapshot in time. The findings and recommendations reflect the information gathered during the assessment and not any changes or modifications made outside of that period.

An audit does not guarantee identification of all potential vulnerabilities. Anastasia Labs makes a best effort assessment to identify all security vulnerabilities and attack vectors.

Assessment overview

From January 3rd, 2024 to March 3rd, 2024, Genius Yield engaged Anastasia Labs to evaluate and conduct a security assessment of its Genius Yield DEX protocol. All code revision was performed following industry best practices.

Phases of code auditing activities include the following:

- Planning – Customer goals are gathered.
- Discovery – Perform code review to identify potential vulnerabilities, weak areas, and exploits.
- Attack – Confirm potential vulnerabilities through testing and perform additional discovery upon new access.
- Reporting – Document all found vulnerabilities.

The engineering team has also conducted a comprehensive review of protocol optimization strategies.

Assessment components

Manual Review

Our manual review process involves a number of different phases. In the first phase, we look for common vulnerabilities and attack vectors that are protocol agnostic. Then in the second phase, we perform a thorough analysis of the code-base and the technical specification of the protocol in which we attempt to identify any points at which the implementation differs from the specification. Finally, in the final phase we perform a review of the on-chain utility and library functions to check that they work as intended and are used only where appropriate.

Code base

Repository

https://github.com/geniusyield/dex-contracts-api

Commit

84347e4a5e6268e3bd66f1b83406b0be341f1074

Files audited

SHA256 Checksum	Files
ed071d7d7a6407dd7af7e79a1b68360b143bdbe9044660b8d93c7a907a0299b0	/src/GeniusYield/OnChain/DEX/PartialOrderNFTV1_1.hs
0495d86ef9eb19054b88a3cf58151cc0bb77680c9ace0d3a55c8b8c13f9a28fb	/src/GeniusYield/OnChain/DEX/PartialOrder.hs
931fe247e9f5e14672cd9257a51bb1f6fe9a3e6f839a04c53d93f34a3b024aa4	/src/GeniusYield/OnChain/DEX/NFT.hs
cd93d3ccc41b3d2fd7762c2bb6f544682301c6b60cab67ba439d150735005e83	/src/GeniusYield/OnChain/DEX/PartialOrderConfig.hs

Finding severity ratings

The following table defines levels of severity and score range that are used throughout the document to assess vulnerability and risk impact.

Level	Severity	Findings
5	Critical	1
4	Major	0
3	Medium	0
2	Minor	1
1	Informational	5

The audit used the Common Vulnerability Scoring System in conjunction with the NVD Calculator to provide a standardised measure for the severity of the identified vulnerabilities. We recognize that many of the metrics considered in CVSS are not relevant for audits of Cardano Smart Contracts; nonetheless, there is still considerable value in adhering to a standard in that it provides more unbiased severity metrics for the findings.

`https://www.first.org/cvss/v3.1/specification-document#Qualitative-Severity-Rating-Scale`

Executive summary

The Genius Yield DEX smart contracts aim to uphold the following attributes:

- No central authority (like a traditional centralized exchange) is needed. Users can safely use the protocol without trusting any authority or even one another.
- Protocol fees can be updated dynamically, however, the changes in flat fees cannot be applied retroactively (so updates to fee parameters do not affect old orders).
- The correctness of orders is enforced by the presence of the order minting policy. This allows off-chain agents to identify all valid orders simply by querying for UTxOs that contain order minting policy NFTs.

Findings

ID-501 Insufficient deserialization of Address Plutus Type

	Level	Severity	Status
	5	Critical	Resolved

Description

The PartialOrderConfiguration spending validator does not enforce that the ‘pocdFeeAddr‘ from the datum is indeed a valid Cardano address. This means that `PartialOrderConfiguration` could be updated such that the ‘pocdFeeAddr‘ field in ‘newDatum‘ has a payment or stake credential where the length is not equal to 28 bytes. This would result in an invalid address according to Cardano Ledger rules. Alternatively, the constructor tag of ‘pocdFeeAddr‘ could be set to not match the expected constructors for the Address PlutusType, because the address field in the actual txOutputs uses the correct constructors there can never be an output with an address equal to ‘pocdFeeAddr‘. In either case, when a user attempts to cancel their partially filled order or they attempt to PCompleteFill or perform any other interaction which requires paying fees to the address their transaction will fail due to ledger rules and thus the funds in their partially filled order UTxO will be stuck forever. Effectively, this means the multisig can be used to permanently lock all user funds in partially filled orders.

Recommendation

Enforce that the `PartialOrderConfiguration` datum deserializes to the exact structure expected by the corresponding Plutarch type `PPartialOrderConfigDatum`.

Resolution

Resolved; rewritten in Plutarch and the update must include a payment to the fee address (which guarantees that the address is valid).

ID-201 Inconsistent continuing datum type

	Level	Severity	Status
	2	Minor	Acknowledged

Description

There is no logic to enforce consistency across the partial order input UTxO datum type and the datum type of the corresponding output. This means an agent can change the datum from inline datum to datum hash at will. This behavior is problematic and can impact the ability of various matching agents to identify the actual datum. The number of orders than can be processed in a transaction is significantly reduced if the orders use datum hashes instead of inline datums, this means a malicious agent can process partial fills and change the datum to datum hashes in order to reduce throughput of the protocol. Also, it can impact the transaction fee and minAda. Furthermore, UTxOs with datum hashes are susceptible to a number of vulnerabilities that those with inline datums are not (for instance the datum size attack), so often it can be safer to enforce that all UTxOs must contain inline datums.

Recommendation

If an order is created with an inline datum, then it should remain as inline datum throughout the partial order fills (and vise-versa if it was created with a datum hash it should remain datum hash throughout its life cycle). Alternatively, enforce that all orders must use inline datums throughout their life-cycle (this should reduce fees and increase protocol throughput as-well).

Resolution

Acknowledged

ID-101 Optimization: Unnecessary Plutarch Level Functions

	Level	Severity	Status
	1	Informational	Acknowledged

Description

Every function in the Plutarch portion of the code-base is using Plutarch level functions. This is wasteful if the arguments are not being used multiple times in the function body.

Recommendation

Prefer Haskell level functions unless arguments are used multiple times or where the function itself hoisting is necessary to reduce script size.

Example 1

Example 2

Resolution

Acknowledged

ID-102 Optimization: Redundant Spending Validator Execution

	Level	Severity	Status
	1	Informational	Acknowledged

Description

Currently the partial order validator logic executes once per partial order UTxO that is matched in the transaction. A lot of the logic is executed redundantly across all the order UTxOs (ie destructuring the script context, getting the reference input, and a number of other components). This severely limits the number of orders that can be matched in a transaction to somewhere around 4-6 orders. If you opt to use the withdraw zero trick or any of the other transaction level validation design patterns you can increase the throughput to allow matching 20+ orders per transaction.

Recommendation

Utilize the withdraw-zero trick to avoid redundant execution of the same logic across multiple validators in the transaction. The design pattern is explained in the following link:

`https://github.com/Anastasia-Labs/design-patterns/blob/main/stake-validator/STAKE-VALIDATOR-TRICK.md`

Resolution

Acknowledged

ID-103 Optimization: Avoid Maybe types in on-chain code

	Level	Severity	Status
	1	Informational	Acknowledged

Description

In Cardano smart contract development, the use of Maybe types (ie `PMaybeData`) is an anti-pattern. They introduce a large amount of bloat and the benefits such types provide in Haskell are almost always completely disregarded in onchain code where the desired behavior is often to fast-fail (or to immediately handle an alternative code branch because the language is strict).

For instance:
`https://github.com/geniusyield/Core/blob/3792f093c09797a03ee3c6cf987dbdf162132dfa/geniusyield-onchain/src/GeniusYield/OnChain/DEX/PartialOrder.hs#L84`

Here, we pay the cost of constructing and pattern matching against this `PMaybeData` type only to immediately fast-fail if the result is not `PJust`. Of-course in such cases we should prefer a fast-fail version `pmustFindOwnInput` that returns the `ownInput` if it exists and calls builtin error otherwise. Even more-so since this function should never fail in the first place as the presence of the `ownInput` is guaranteed by the `ScriptPurpose`.

Recommendation

Avoid using `PMaybeData` in onchain code.

Resolution

Acknowledged

ID-104 Optimization: Strict vs lazy boolean operations

Level	Severity	Status
1	Informational	Acknowledged

Description

When fast-fail behavior is desired you should prefer strict boolean operations since the lazy variants introduce wasteful delay and force operations. For instance in the following code:

`https://github.com/geniusyield/Core/blob/3792f093c09797a03ee3c6cf987dbdf162132dfa/geniusyield-onchain/src/GeniusYield/OnChain/DEX/PartialOrderNFT.hs#L119-L120`

It makes sense to use the lazy and `#&&` operator for the first condition `hasUtxoConsumed`; however, for the remaining conditions (which will only execute if `hasUtxoConsumed` is true) the strict variant `pand'` should be used.

Recommendation

Use the strict and operator `pand'` where appropriate.

Resolution

Acknowledged

ID-105 Optimization: Prefer consolidated fast-fail logic over pguardC

	Level	Severity	Status
	1	Informational	Acknowledged

Description

Currently, the fast-fail logic is separated into a number of `pguardC` calls. Each of these calls introduces a number of delays, forces and `builtinIfThenElse` calls into the UPLC. This introduces a lot of wasted computation.

Recommendation

Ideally, these should be consolidated into a chain of strict `builtinIfThenElse` (ie by using `pand'`) where only the final if in the chain is lazy (the one containing the branch with error).

Resolution

Acknowledged