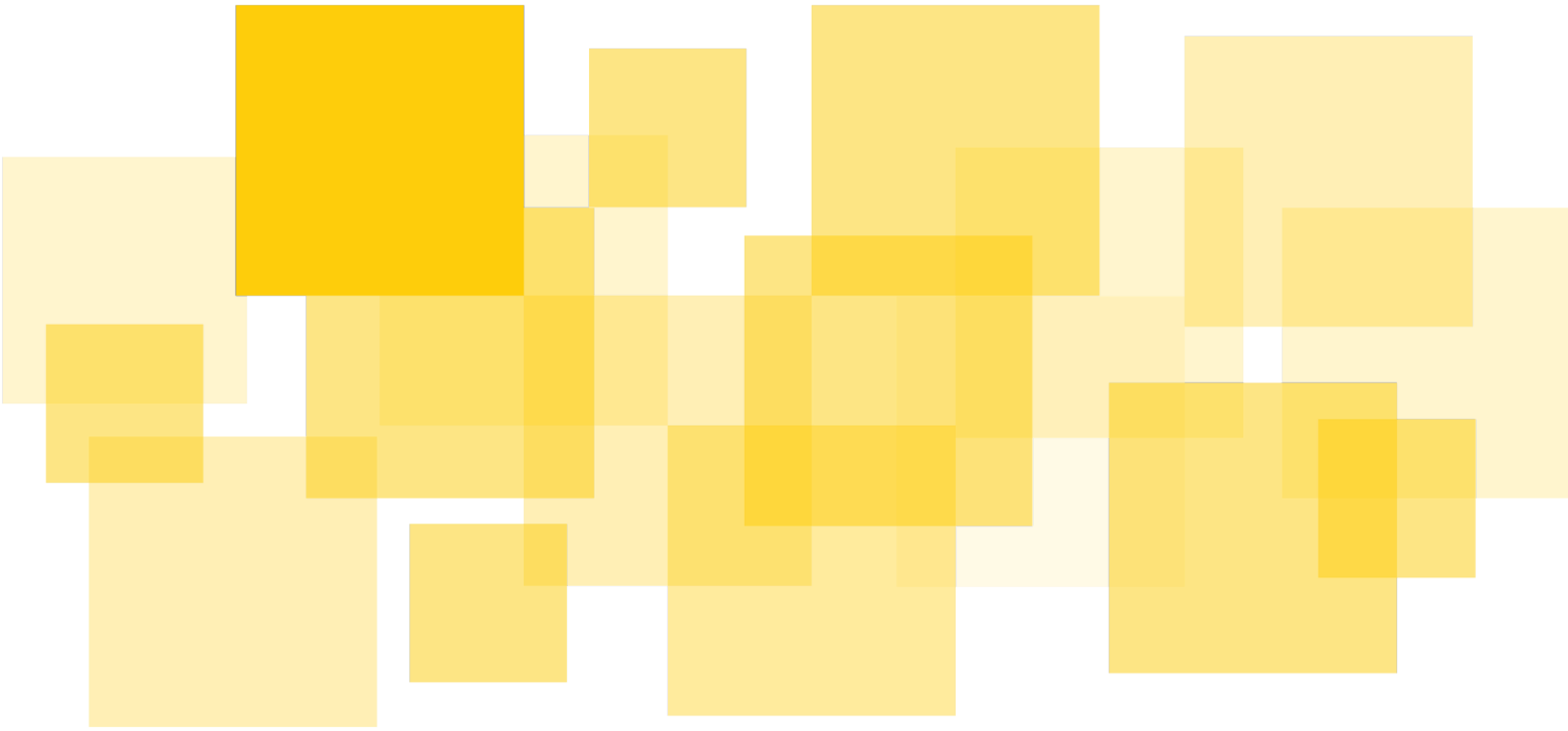
Security Audit Report

SundaeSwap V1 Contracts

January 17th, 2022



Prepared for SundaeSwap Labs, Inc by

runtime verification

Table of Contents

Executive Summary

Runtime Verification, Inc. acted as security code auditor on the SundaeSwap V1 smart contract. The review was conducted from November 04 to December 02, 2021. A follow-up code audit for the code revision, which is meant to address the issues found in the previous code review, was also conducted from December 03 to December 10, 2021.

SundaeSwap Labs also engaged Runtime Verification early as a design advisor in the process of designing V1 of the SundaeSwap smart contract. Runtime Verification has been reviewing the design of these new contracts, and given feedback from a security perspective. This has led to several iterations of designs of liquidity pool and scooper.

The purpose of this code review has been to ensure that the system design has been faithfully implemented and any other implementation issues are identified and addressed. This engagement found several critical issues which have been addressed by SundaeSwap Labs.

We highly recommend that teams reach out to security auditors early in the process of development, rather than at the end. In the case of SundaeSwap Labs, we believe that we have been able to address several issues that would have been harder to resolve later, and that the final system has improved as a result.

Introduction

Runtime Verification engaged in two main audits for SundaeSwap Labs: the code audit and a follow-up audit.

Scope

The code audit phase focuses on reviewing one code repository assisted with an informal design documentation. The informal design documentation is referenced to facilitate our understanding of the design and implementation and is ***not*** part of this engagement. The code review is conducted on *source code* of the commit hash **c56dbd7ae2467a0142c8219670e135ab66376e44** of the contracts repo. The following source code files in this commit hash are included in the review:

- onchain/Sundae/ShallowData.hs
- onchain/Sundae/Utilities.hs
- onchain/Sundae/Contracts/Common.hs
- onchain/Sundae/Contracts/Factory.hs
- onchain/Sundae/Contracts/Mints.hs
- onchain/Sundae/Contracts/Others.hs
- onchain/Sundae/Contracts/Pool.hs

The review is limited to only the above source files which are the onchain code of the contracts. Any other artifacts such as off-chain and deployment and upgrade scripts of the repo are not part of this review.

The purpose of the follow-up code audit is to make sure that all the issues found in the code audit phase have been addressed. The follow-up audit results in the above source code files in the commit hash **b7dde6915859edc40b502919dcd56b3305b29508** of the same contracts repo. In the follow-up audit phase, the auditor reviewed the pull requests for the issues found by the auditor in the code audit phase (findings A01-A12 and B01-B03). The auditor also helped to review pull requests to address the performance improvements suggested by a third party (findings P01, P02, P03, and P04) and an issue found by an external user of the SundaeSwap testnet (finding E01).

Protocol Configurations

The following initial protocol settings specified by SundaeSwap Labs are used to bootstrap the smart contracts:

1. Protocol boot UTXO: 213c3838a997b9e1a48334097acb6257099bb8001593c22d583afca429cd5bbf#1

2. Treasury boot UTXO: 213c3838a997b9e1a48334097acb6257099bb8001593c22d583afca429cd5bbf#2
3. Initial set of two scoopers: addr1vxw3ew65ltegfawjvt6erv0eyqdpskx7z4g40kk5nuugr3q5w2w2x whose public key hash is 9d1cbb54faf284f5d262f591b1f9201a1858de155157dad49f3881c4 and addr1v955h3sp07whffwek0hnw76zh8lyjeaqf7ccgj2eq4lntwcmgnrwr whose public key hash 694bc6017f9d74a5d9b3ef377b42b9fe4967a04fb1844959057f35bb
4. The protocol upgrade time lock period is 604800 seconds which is equivalent to 7 days.
5. The scooper fee time lock period is 5 weeks.
6. The protocol upgrade authentication policy id: 2739b84594eb8c5745b67795d07c67c9bdd1dbd52cdad176f5d33115
7. The protocol upgrade authentication token name: 696e746572696d2d676f7665726e616e6365
8. Entropy from Random.org using the factory script hash as a seed.

As part of doing so, we modified some auxiliary code with commit hash **8b1fc79fb4cdbaaa5e9d4749b43d556c34c146f8**. These changes have no impact on the smart contracts but make it easier to verify these scripts hashes.

Runtime Verification independently verified that the above settings and procedure resulted in the following script hashes:

1. Factory boot policy id: e8a447d4e19016ca2aa74d20b4c4de87adb1f21dfb5493bf2d7281a6
2. Factory NFT token name: 666163746f7279
3. Treasury boot policy id: 8db8893ffcc12744fb5fc0c6c1aed548f76873cf7e07d2dc7d41a259
4. Treasury NFT token name: 7472656173757279
5. Sundae policy id: 9a9693a9a37912a5097918f97918d15240c92ab729a0b7c4aa144d77
6. Sundae token name: 53554e444145
7. Pool policy id: 0029cb7c88c7567b63d1a512c0ed626aa169688ec980730c0473b913
8. Factory script hash: addr1w82z6yrftsxz77el0ce2q4vuspcym2x0xgpgneurrwvasfge778fd
9. Pool script hash: addr1w9qzpelu9hn45pefc0xr4ac4kdxeswq7pndul2vuj59u8tqaxdznu
10. Treasury script hash: addr1w9742z4fewans7ry6cjp95pc4ecv7y54cx298lp5qfw7s9gv8ukrj
11. Escrow script hash: addr1wxaptpmxcxawvr3pzlhgnpmzz3ql43n2tc8mn3av5kx0yzs09tqh8
12. Scooper Fee holder script hash: addr1w9jx45flh83z6wuqypyash54mszwmdj8r64fydafxtfc6jgrw4rm3
13. Gift script hash: addr1wyje29slhjgulj7ww3vxxpgqmfjwcx3gdn4mkchlwdn9djqz5huyy
14. Proposal script hash: addr1wykfq3na0gduva96y703t4wlusy9dqr6shdg3l8syzn5x7gjj9z4v
15. Dead factory script hash: addr1wxkzcjh3k7sm4vmvppuyqdagc38j7yunj5tltrzn0ym3xasd5dety
16. Dead pool script hash: addr1w929tfaltf6s9nvl7z3cj7uv9hul4747649v3wa3j384k0gntgcz3

Assumptions

All the code reviews are based on the following assumptions and trust model:

1. The external governance contracts are assumed to be safe and trustworthy. We assume that the governance system generates an UTXO containing the protocol upgrade authentication token and the upgrade data and sends the generated UTXO to be locked by the `proposalContract`. The asset class of the authentication token is a configuration to the contracts.
2. The design of the contracts assume a set of trusted scoopers which are voted in by the community through the governance system. The scoopers are trusted to be honest.
3. This code repository is based on the V1 of Plutus API. We also assume the correctness of the Plutus platform. In Particular, we assume that the Plutus Core compiler works correctly by compiling the Haskell code to the Plutus Core code.

Methodology

We conduct the code review manually. Although manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have followed the following approaches to make our review as thorough as possible. First, we rigorously reasoned about the business logic of the contracts, validating security-critical properties to ensure the absence of loopholes in the business logic. Second, we carefully modeled the design in pseudo code so that we can find details which are missing from the specification. The two steps were done in our design advisory engagement. Finally, we modeled the dependency (token dependency for transactions and token minting) among the contracts; tried out all possible combinations of UTXOs to construct desired and undesired transactions; checked the following categories of vulnerabilities for each possible transaction:

1. Token related issues. Tokens are assets and are the means to share data (together with the datum) and to achieve access control in Plutus contracts. So the correctness of minting and usage of the tokens and their datum is paramount for the correct functioning of Plutus contracts.
2. Plutus specific implementation issues. There are semantic gaps between the source code language (Haskell) and the onchain assembly code (Plutus Core). It is easy for developers to ignore issues that come inherently in the Plutus platform.
3. Constant Product Market Maker(CPMM) specific issues. The CPMM formula involves a specific way of calculating the prices and tokens.

We considered the following attack surfaces for token related issues:

1. Fake token attacks
2. Unsound minting policies
3. Side minting attacks
4. Token uniqueness attacks

5. Violation of single transaction execution
6. Contract clone attacks
7. Deadly transaction repetition

All the issues have been reported to and are either acknowledged or addressed by SundaeSwap Labs.

Severity Classification

The following bug severity classification system is proposed by SundaeSwap Labs and is used in the report:

- P1: Highest priority; loss of funds, deadlock of funds, hijack of protocol, arbitrary minting of tokens, etc.
- P2: Severe disruption, DDOS, etc. No funds are technically lost, but makes the protocol unusable for long periods of time
- P3: Inconvenient, or non-intuitive behavior of the protocol; loss of funds through users fault, but where the mistake is very easy to make; etc.
- P4: Code organization, performance, clarity; no major / logical impact on the protocol, but technically changes the behavior
- P5: Purely documentation / cosmetic, like wording in comments or error messages, etc.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive.The possibility of human error in the manual review process is real. We recommend always seeking multiple opinions and audits.

List of Findings

The following list of findings consists of issues identified by the auditor from Runtime Verification (A01-A12, B01-B03) as well as issues identified by others (P01-P04 and E01). If the issue is identified by others other than the auditor, its identifier is specified explicitly in the “Identified By” section. Also take note that the recommendations in P01 to P04 are suggested by others, not by Runtime Verification. We include the issues identified by others in this report because the auditor helped to review the pull requests for such issues in the follow-up audit phase.

A01. Minting of unlimited number of sundae tokens

Sundae tokens (token name SUNDAE) are the native currency of the SundaeSwap contracts. Sundae holders can decide community events such as upgrading current contracts through voting in the governance system.

However, unlimited SUNDAEs can be minted by consuming the treasury token (an NFT locked by the `treasuryContract`) with the `SpendIntoTreasury` redeemer. This is because the `sundaeMintingScript`, which controls how Sundaes are minted or burned, does not check how many Sundaes have been minted. It only requires the treasury token to be present in one of the pending transaction’s outputs (thus also in the transaction’s input because there is only one treasury token in the system).

Severity: P1

Recommendation

The specification of SundaeSwap contracts requires that only a maximum number of Sundaes can be minted. The SundaeSwap team will mint the maximum number of Sundaes when spending the treasury token into the `treasuryContract`. The recommendation here is to disable token minting in the code related to the `SpendIntoTreasury` redeemer in `treasuryContract`.

Status: Addressed by client. See Pull Request 295.

A02. Accumulation of rounding errors when exchanging old liquidity tokens for new liquidity tokens

Description

The `scaleInteger` function in *onchain/Sundae/Contracts/Common.hs* uses the `round` function provided by *PlutusTx.Ration.hs* to convert the Rational value to an Integer value. However, the `round` function rounds up when the fraction part is greater than or equal to 0.5 and rounds down otherwise. It could be possible that if the fraction part is greater than or equal to 0.5, the contract returns 1 more token to the user and this effect could accumulate and may cause the last user to receive far less than expected amount.

Severity: P1

Recommendation

Use the floor function to do the rounding in `scaleInteger`.

Status: Addressed by client. See Pull Request 301.

A03. Any tokens with the currency symbol as the hash of poolMintingPolicyContract can be minted

The `factoryScript`, when making a script upgrade proposal, only checks that tokens whose currency symbol is the hash of `factoryBootMintingScript` cannot be minted. However, the specification requires that no tokens can be minted in such transactions.

Severity: P1

Recommendation

Modify the code to disable minting of any token.

Status: Addressed by client. See Pull Request 299.

A04. Old script hashes shall never be reused when upgrading factory or treasury

The proposal token generated by the governance system is used to upgrade the `factoryContract` and `treasuryContract`. The proposal token and its datum is locked indefinitely by the `proposalContract`. Although there is a check that the proposal token has a field called `proposedOldFactory` for `factoryContract` upgrade or `proposedOldTreasury` for `treasuryContract` matches the current script hash. However, it is still possible in future upgrades that the governance decides to downgrade to an old version of the contract and decides to reuse the old script hash. This creates a loop in the upgrade history, and an attacker can upgrade the contracts repeatedly using the proposals in the upgrade history loop.

Severity: P2

Recommendation:

Never reuse old hashes for the contracts when downgrading the contract.

Status: Acknowledged. This is an operation issue, no code change is needed.

A05. More scooper fees could be collected with multiple scoopers public key hashes

The scooper fee contract wants to send fees in excess of the maximum to the gift contract, but the way it works for multiple fee inputs is not intuitive. It counts the total amount of fees being redeemed, not the amount per input. So if multiple public key hashes are provided in the signatories, the total fees collected is much larger but only a small amount is paid to the gift contract.

Severity: P1

Recommendation

Multiply the maximum amount with the number of inputs and subtract this amount from the total scooper fees. After discussion, SundaeSwap decided to reward all scooper fees to scoopers, not paying to the treasury.

Status: Addressed by client. See Pull Request 329.

A06. Scooper could redeem more than the maximum number of scooper rewards

The pool contract does not check the positivity of `scoopFee` in `EscrowDatum` and pays the sum of all the scoop fee to `scooperFeeContract`. If an escrow input with negative scoopFee is included, then much less than the actual scoop fee would be paid to the scooper fee contract and the lesser amount is paid to the scooper directly. After that the scooper can redeem the scoop fee from the scooper fee contract. In total the scooper can gain all the fees and circumvent the constraint that only a maximum scooper fee shall be redeemed by scooper.

Severity: P1

Recommendation

Add checks to ensure that only positive `scoopFee` is used in `EscrowDatum` and the Escrow input contains enough to cover the `scoopFee`.

Status: Addressed by client. See Pull Request 327.

A07. Potential unauthorized script upgrade

Line 315 at the `proposalContract` only checks that the upgrade authentication token is present but does not check how many tokens are in the UTXO. This line should also check that the number of authentication tokens is 1. If let's say two tokens are included in the proposal UTXO, an attacker can pay one of the tokens to itself and modify the datum and pay it back into the `proposalContract`. So he can achieve an unauthenticated upgrade to the contracts.

Although it is considered a bug of the governance system if more than 1 tokens are given to the UTXO. But to prevent the ripple effect of such bugs in the governance system, we would better have the check here.

Severity: P2

Recommendation

Add the check that only 1 upgrade authentication token is present in the UTXO.

Status: Addressed by client. See Pull Request 316.

A08. Assets in escrow contract can be stolen when upgrading pool

The UTXOs locked in `escrowContract` can be spent when a pool token is present. It does not restrain how the assets in UTXO can be spent. The pool token can be spent when upgrading a pool, which has checks to ensure that there is an update to the pool token. So when upgrading the pool, an attacker can also spend the UTXO in `escrowContract` so as to steal the assets.

Severity: P1

Recommendation

Add a check that when the pool token is spent into the `deadPoolContract`, then there shall be no escrow inputs.

Status: Addressed by client. See Pull Request 326.

A09. Scooper fees cannot be redeemed into treasury

The `treasuryContract` only allows Ada, Sundae to be locked together with the treasury token. The attacker can pay 1 garbage token together with the scooper fee (Ada) into the `giftContract` when he is redeeming the excessive scooper rewards. This 1 garbage token makes all the scooper fee cannot be paid into treasury and indefinitely locked in the `giftContract` (until a treasury upgrade is done).

Severity: P2

Recommendation

When paying the assets in `giftContract` into treasury, only Ada and Sundae assets are paid into, other tokens are spent back into the `giftContract`.

Status: Addressed by client. See Pull Request 328.

A10. Integer division is used multiple times in computation of token amount

The code related to the calculation of the amount of the other tokens to return to customers uses integer division multiple times. However, the rounding effect of integer division can be magnified by multiplication with a large integer and results in very inaccurate token amounts.

Because of the rounding effect of integer division, we shall do a division only once to make sure the effect of rounding cannot be exacerbated (by multiplying the result of division with another large integer). The division shall be done in the last step to compute the amount of tokens. For example, the takes variable at line 289 shall be calculated with a single division in the final computation, as the following:

```
(b*gives*(denominator swapFees - numerator swapFees)) `divide`
(a * denominator swapFees + gives * (denominator swapFees -
numerator swapFees))
```

Severity: P1

Recommendation

Change the program to only allow 1 single division in the computation of token amount.

Status: Addressed by client. See Pull Request 335.

A11. Rounding error is exacerbated in DepositSingle operation in pool contract

In this operation, the swap fee shall be subtracted from the user deposit amount. However, in the implementation, the swap fee is firstly calculated as an integer value with an integer division and then the swap fee is subtracted from the user deposits. The result of subtraction is then multiplied by another integer and then used to compute the quantity passed to the square root function. With the above process, a much larger number is passed to the square root function due to the integer rounding error.

Severity: P2

Recommendation

The square root function takes a rational as parameter, we can calculate the quantity as a rational number and pass the rational number to the square root function. This way it prevents the exacerbating of the integer rounding error.

Status: Addressed by client. See Pull Request 338.

A12. Assets in escrow contract can be stolen when redeeming liquidity tokens

The UTXOs locked in `escrowContract` can be spent when a pool token is present. It does not restrain how the assets in UTXO can be spent. The (old) pool token is locked by the `deadPoolContract` after the pool upgrade. The `deadPoolContract` allows holders of old liquidity tokens to redeem for the new liquidity tokens for the new pool. An attacker who holds some old liquidity tokens can also spend the UTXO in `escrowContract` so as to steal the assets when redeeming against the `deadPoolContract`.

Severity: P1

Recommendation

Disallow escrow inputs from the `escrowContract` in the `deadPoolContract` when redeeming liquidity tokens.

Status: Addressed by client. See Pull Request 349.

B01. Use of If-then-else for the or operator in poolMintingPolicyContract

The boolean operators `&&` and `||` are strict in Plutus; they do not short-circuit. This is generally not so much of an issue for `&&`, because most transactions in practice will pass script validation and therefore all branches of the conjunction will have to evaluate. However, when we're evaluating a passing transaction against a contract of the form p || q || ... || r, short-circuiting evaluation can save resources.

Severity: P4

Recommendation

Use `If then else` to replace the `||` operators in Line 132-134 at onchain/Sundae/Contracts/Mints.hs.

Status: Addressed by client. See Pull Request 315.

B02. Unconstrained usage of scooper token makes tracing of scoopers hard

A legitimate scooper operator can generate many scooper tokens and send them to anyone. Then the scooper token holder can do scoop operation by falsely declaring that he is scooping on behalf of any legitimate scooper operator by specifying a legitimate scooper operator's public key. This makes tracing of bad scoopers very hard. The tracing of bad scoopers is part of the offline activity by SundaeSwap to determine and scrape bad scoopers (through voting in the governance system).

Severity: P3

Recommendation

Runtime Verification recommends to put the public key hash as part of the scooper token name and use it to validate the transaction. However, SundaeSwap thinks the recommendation has a huge performance impact on the transaction throughput and also thinks the issue shall not be deemed as a vulnerability and chooses to not implement this recommendation in the contract.

Instead, SundaeSwap will set expectations for scooper operators through operational measures. The expectation is that scooper operators shall not share the scooper tokens with others and, if any dishonest use of the scooper token by anyone is detected, the scooper operator who issued the scooper token shall be deemed dishonest.

Status: Acknowledged.

B03. Redundant checks in factory contract

At the start of the factory contract, if the redeemer is not `UpgradeFactory`, we check that no tokens were added or removed. However, it also checks that condition in redeemer `MakeProposal` and `IssueScooperLicense` later in the code.

Severity: P4

Recommendation

Remove the redundant checks in the two redeemers code branches.

Status: Addressed by client. See Pull Request 350.

P01. Avoid higher-order functions and extra list traversals

Function `mergeListByKey` traverses its argument twice, when only one traversal is needed. `sansRider` calls `Map.insert`, which is relatively inefficient compared to calling `Map.fromList`.

Identified by: IOG

Severity: P4

Recommendation

Rewrite `mergeListByKey` with explicit recursion. Rewrite `sansRider` to do more efficient map operations.

Status: Addressed by client. See Pull Request 308.

P02. Optimize `atLeastOne` uses, script address checks, `flattenValue` in Pool script

- Uses of unoptimized native functions `atLeastOne`, `findOwnInput`, and `flattenValue` can be replaced with more efficient custom implementations.
- Address comparison performance can be improved.
- Temporary variables can be introduced to avoid redundant computation.

Identified by: IOG

Severity: P4

Recommendation

- Implement custom implementations of native functions.
- Optimize address equality comparison.
- Introduce variables.

Status: Addressed by client. See Pull Request 309.

P03. Avoid redundant computation in `doEscrows`

`userGives $$ CoinA` is needlessly computed multiple times. fee variables don't need to be strict as they are only used once.

Identified by: IOG

Severity: P4

Recommendation

Define a new variable `giveCoinA`. Remove unnecessary strictness annotations**.**

Status: Addressed by client. See Pull Request 307.

P04. Rewrite `rawDatumOf` function

Native function `findDatum` called from `rawDatumOf` incurs unnecessary memory and CPU usage that can be improved by using an explicit loop.

Identified by: IOG

Severity: P4

Recommendation

Rewrite `rawDatumOf` in *Sundae.Utilities* to avoid the use of higher order functions and closures in native function `findDatum`.

Status: Addressed by client. See Pull Request 306.

E01. Less escrow riders are returned when multiple escrows from the same trader are scooped in a transaction

An escrow input must contain the correct amount of tokens regarding the type of the order and a 2 ADA rider. The rider shall be spent back to the trader. However, if multiple escrow inputs (`n>=2`) from the same trader are scooped in the same transaction, only a single 2 ADA rider is required to be spent back to the trader. This results in `2*(n-1)` ADAs taken by the scooper from the trader.

Identified By: Rodgrig0v

Severity: P1

Recommendation

Track the number of orders from the same trader and ensure that the correct rider amount gets spent to the trader.

Status: Addressed by client with caveat. See Pull Request 340.

Caveat: Ideally we would ensure that a trader cannot lose money to a mistake by providing extra tokens. Due to severe performance implications, SundaeSwap does not enforce this check in the contract. If a trader gives more tokens than necessary for the escrow operation plus the rider, the extra tokens may be taken by the scooper.