Technical Review of MinSwap

Smart Contract Verification Team

Contents

Chapter 1

Executive Summary and Scope

This report is presented without warranty or guaranty of any type. This report lists the most salient concerns that have so far become apparent to Tweag after a partial inspection of the engineering work. Corrections, such as the cancellation of incorrectly reported issues, may arise. Therefore Tweag advises against making any business decision or other decision based on this report.

Tweag does not recommend for or against the use of any work or supplier referenced in this report. This report focuses on the technical implementation provided by the project's contractors and subcontractors, based on their information, and is not meant to assess the concept, mathematical validity, or business validity of MinSwap's product. This report does not assess the implementation regarding financial viability nor suitability for any purpose.

Scope and Methodology

Tweag looks exclusively at the on-chain validation code provided by MinSwap. This excludes all the frontend files and any problems contained therein. Tweag manually inspected the code contained in the respective files and attempted to locate potential problems in one of these categories:

a) Unclear or wrong specifications that might allow for fringe behavior.

b) Implementation that does not abide by its specification.

c) Vulnerabilities an attacker could exploit if the code were deployed as-is, including:

- race conditions or denial-of-service attacks blocking other users from using the contract,
- incorrect dust collection and arithmetic calculations (including due to overflow or underflow),
- incorrect minting, burning, locking, and allocation of tokens,
- authorization issues,

d) General code quality comments and minor issues that are not exploitable.

Where applicable, Tweag will provide a recommendation for addressing the relevant issue.

Chapter 2

Audit

2.1 Methodology

Tweag analyzed the validator scripts comprising the MinSwap protocol as of commit `16255227` of the repository[1] shared with us. The names of the files considered in this audit and their `sha256sum` are listed in Table 2.1. We had to perform minor modifications to MinSwap's code to export certain names and derive some extra instances, and the corresponding patch is listed in Appendix A.1.

Our analysis is based on the original documentation provided by MinSwap, on the expanded documentation that came on a later commit (`3eaf2edf`), and on Slack conversations with MinSwap. The relevant documentation files are listed in Table 2.2 and their contents will be referred to as *the specification* of the protocol.

Tweag is delivering a custom version of our `cooked-validators` library adapted for MinSwap's validators, due to their custom PAB requirements. Tweag *does not* commit to maintaining this custom version of `cooked-validators` and reserves the right to merge none or any number of these modifications to the library's `main` branch. For client's convenience, we also package the snapshot of `cooked-validators` with our other deliverables.

The MinSwap protocol is a collection of validator scripts and minting policies that enable creating constant product pools and processing Uniswap-like orders over these pools. The notion of a *batcher* is central to MinSwap's protocol: these are entities that receive permission to apply open orders against pools. This permission is granted by means of a *license token*, which has an expiration date.

Tweag *did not* check the correctness nor precision of the pool price calculations since this was absent from the specification. Checking the contract for financial viability falls out of the scope of this review.

Because MinSwap built their off-chain code in Go, we began by writing our own transaction generation code with our `cooked-validators` library, in Haskell. Throughout our analysis, we assume that (A) there exists at least one batcher and (B) the holder of the *owner* token is honest (implying the token cannot be stolen, its holder's computer cannot be hacked, and so on). Although MinSwap's code is written assuming that batchers are honest, we believe that, given the absence of detail on how their licenses are distributed and the lack of accountability in case a batcher misbehaves (or gets their license stolen), this is too important to ignore.

2.2 Findings

Table 2.3 lists our concerns with the current MinSwap's code base based on our partial exploration during a limited period of time. Throughout the rest of this section, we will detail each of our findings.

2.2.1 Vulnerabilities

2.2.1.1 ■ Unauthorized Redeeming of Open Orders

Severity: *Critical*

[1] `https://github.com/minswap/minswap-dex-tweag`

sha256sum	File Name
f6aa62...688edb	src/Minswap/ConstantProductPoolNFT/Utils.hs
b8f6f5...a933ca	src/Minswap/ConstantProductPoolNFT/OnChain.hs
c1d523...3fc89d	src/Minswap/BatchOrder/Types.hs
91c490...e4b783	src/Minswap/BatchOrder/OnChain.hs
ffa880...ad1758	src/Minswap/ConstantProductPool/Utils.hs
d13a9e...17bd89	src/Minswap/ConstantProductPool/Types.hs
0781b3...dff5e9	src/Minswap/ConstantProductPool/OnChain.hs
509d1e...cd1a79	src/Minswap/ConstantProductFactory/Types.hs
37a9c4...a5e927	src/Minswap/ConstantProductFactory/OnChain.hs
f06d56...2354d9	src/Minswap/Types/Coin.hs
54ad61...a1b605	src/Minswap/MinswapCLI.hs
d72ab4...490ad0	src/Minswap/Spooky/TypedSpookyContexts.hs
b86f40...8413d2	src/Minswap/Spooky/Typed.hs
2201c9...9dbc77	src/Minswap/Spooky/UntypedSpookyContexts.hs
83f50f...9d13ec	src/Minswap/Spooky/Untyped.hs
f40752...a91218	src/Minswap/Utils/OnChainUtils.hs
70ff43...8d5935	src/Minswap/ConstantProductLiquidity/OnChain.hs

Table 2.1: *Files analyzed modulo the patches described in Appendix A.1*

sha256sum	File Name
12812a...b017dd	docs/overview.md
f515ba...1d717e	docs/tx-spec.md

Table 2.2: *Documentation files and their `sha256sum` as of commit `3eaf2edf`*

Severity	Section	Summary
Critical	2.2.1.1	Unauthorized Redeeming of Open Orders
Critical	2.2.1.2	LP Tokens Can Be Duplicated
Critical	2.2.1.3	Unauthorized Hijacking of Pools Funds
High	2.2.3.1	Batchers Can Choose Pools
High	2.2.3.2	Batchers Licenses Cannot be Revoked
Medium	2.2.2.1	Batchers Can Choose Batching Order
Medium	2.2.3.3	Assumptions on Batcher's Licenses Distribution
Medium	2.2.4.1	Reliance on Indexes Into ScriptContexts' `txInputs` and `txOutputs`
Medium	2.2.4.2	Duplication of ScriptContext Definition
Medium	2.2.4.4	Protocol Specification Lacking
Low	2.2.2.2	Batcher Is Not Allowed to Apply Their Own Orders
Low	2.2.3.4	Pools Cannot Be Closed
Low	2.2.4.3	Large Refactoring Opportunities

TABLE 2.3: *Table of findings*

Open orders can be redeemed by anybody with the **`ApplyOrder`** redeemer as long as the attacker also redeems one output belonging to the address of the `poolCode` validator. This can be easily done by redeeming any pool with profit-sharing enabled with **`WithdrawLiquidity`**. This way attackers can steal the funds locked in open orders.

The ideal solution is to use `txInfoRedeemers` from **`Plutus.V2.Ledger.Contexts.TxInfo`**, in the `batchOrderCode` validator. This way, the `batchOrderCode` validator can ensure that a transaction also redeems a **`PoolDatum`** with the **`ApplyPool`** redeemer. However, this approach *will not* work in the short term: anything **`Plutus.V2`** is unfortunately not supported on-chain just yet. If MinSwap needs a fix using **`Plutus.V1`**, a possible workaround could be to restrict the withdraw liquidity endpoint to be accessible to transactions signed by the "owner" only. This workaround relies on the fact that (A) "owner" is honest and (B) there is only one publicly accessible redeemer to redeem a **`PoolDatum`**, which is to apply an order. Any change in either of these conditions would require a re-evaluation of this recommendation.

2.2.1.2 Critical LP Tokens Can Be Duplicated

Severity: Critical

The minting policy `lpCode` from **`Minswap.ConstantProductLiquidity.OnChain`** is too loose. It allows LP tokens of a pool to be minted by any transaction containing a single output belonging to any script, as long as the output contains a pool NFT. In this case, an attacker can redeem a pool with **`WithdrawLiquidity`** and mint an arbitrary amount of that pool's LP tokens in the process. Next, this attacker is equipped to empty the pool in question by sending a **`Withdraw`** order. The only precondition is that the pool under attack has profit-sharing turned on.

Similarly to issue 2.2.1.1, the ideal solution would rely on the **`ScriptContext`** from **`Plutus.V2.Ledger.Contexts`** and further restrict the minting of LP tokens to transactions that redeem a pool with **`ApplyPool`**. Since that is not possible, the suggestions A and B in issue 2.2.1.1 will mitigate unrestricted minting of LP to-

kens. Orthogonally, one could strengthen the minting policy of LP tokens to only allow minting when a transaction also consumes a UTxO containing a batcher's license.

2.2.1.3 Unauthorized Hijacking of Pools Funds

Severity: Critical

The pools are vulnerable to a datum-hijacking attack. The attack starts by creating a stealer script with a datum and redeemer combination that is isomorphic to the `poolCode` from **`ConstantProductPool.OnChain`**. Next, the attacker redeems a legitimate pool validator, but it "pays to" the stealer script. Because the `ownOutput` is selected without regards to the address:

```
ownOutput :: TxOut
!ownOutput = case [o | o <- txOutputs, hasPoolNFT (txOutValue o) ppNftSymbol] of
               [o] -> o
```

the `ownOutput` will contain the output that pays to the attacker's script. Moreover, because it has an isomorphic datum to **`PoolDatum`**, the `outputPoolDatum` gets decoded properly.

```
outputPoolDatum = mustFindScriptDatum @PoolDatum ownOutput info
```

This is a severe vulnerability and can be executed on any of the three redeemers from `poolCode`:

- **`WithdrawLiquidity`**: Most concerning since anybody can execute it as long as the pool under attack has profit-sharing turned on.
- **`ApplyPool`**: Generally concerning since batchers can execute this attack, emptying a pool.
- **`UpdateFeeTo`**: Although vulnerable to this attack, the attacker would have to be holding the "owner" token, which contradicts our assumption of the holder of the "owner" token being honest.

We recommend using `getContinuingOutputs` from **`Ledger.Contexts`** to select an output that is the evolution of the current script datum. This vulnerability is closely related to our general concern in 2.2.4.2. By duplicating and modifying **`ScriptContext`** from **`Plutus.V1.Ledger.Contexts`**, it is easy to overlook the necessity to use the `getContinuingOutputs` function.

2.2.2 Implementation Bugs

2.2.2.1 Batchers Can Choose Batching Order

Severity: Medium

A batcher that decides to not rely on the MinSwap application backend can choose a different order of buy/sell/swap orders than the expected chronological order. This is categorized as a bug because the specification mentions that orders need to be processed in *fair* order, but the on-chain code does not enforce that. Hence, the code does not abide by the specification. We think it would be too difficult to enforce this constraint at the on-chain level and, hence, suggest to loosen the specification and to design a way to mitigate batchers that do not respect this property.

In particular, because the licenses cannot be revoked (issue 2.2.3.2), a batcher's misbehavior cannot be stopped until their license expires. Depending on the validity interval of the licenses, this can be a problem. The discussion on issue 2.2.3.2 expands on this concern.

2.2.2.2 Batcher Is Not Allowed to Apply Their Own Orders

Severity: Low

As it currently stands, a batcher is not allowed to place their own orders. While one can imagine this to be a good thing (for instance, due to the possible conflict of interests of batchers), it is unclear from the specification whether this is the intended behavior. In `validateApplyPool` we have:

```
userInputs :: [TxInInfo]
!userInputs = [i | i <- txInputs, isUserInput i]

isUserInput :: TxInInfo -> Bool
isUserInput txIn =
  txInInfoOutRef txIn /= txInInfoOutRef ownInput
    && txOutAddress (txInInfoResolved txIn) /= batcherAddress
```

Hence, by construction, an order created by a batcher cannot be applied by the same batcher. Strengthening the specification or using some of the simplifications listed in issue 2.2.4.1 would address this.

2.2.3 Unclear Specification

2.2.3.1 Batchers Can Choose Pools

Severity: High

Whenever a batcher picks an open order other than a *withdraw*, they can select whichever pool they want to execute that order against. Moreover, they can create a custom pool just to execute said order. The specification does not mention whether this is intended or not. We believe that orders could contain some information about which pool the creator of the order wants it executed against.

2.2.3.2 Batchers Licenses Cannot be Revoked

Severity: High

A batcher's license is final and irrevocable until its deadline is reached. Because of the high permissions and trust deposited on batchers, and depending on how these wallets are selected, the incentive for misbehavior can be high. Additionally, it is to be expected that the private keys to those wallets will be sought after by malicious third parties.

We recommend weighting out some options to make licenses non-transferable and revocable. One option is the representation of licenses through an additional validator. Instances of said validator would only be allowed to be created by the "owner", but could be redeemed either by the "owner" for canceling them, or by its respective batcher for applying orders to a pool. This would enable the "owner" to immediately revoke misbehaving batchers.

A less centralized design might include the address of the batcher in the license itself, and also include the set of undesirable batchers within the orders (for example, using a bloom filter). This way, order creators could specify which batchers *cannot* redeem their orders.

2.2.3.3 Assumptions on Batcher's Licenses Distribution

Severity: Medium

Precisely because of the issues discussed in issue 2.2.3.2, it is important to document how these licenses are to be distributed and renewed. Currently, this information is absent from the specification.

2.2.3.4 Pools Cannot Be Closed

Severity: Low

Pools can be created by anyone, but can never be closed. This means that whoever creates the pool will lose their `minAdaPerUTxO` forever. Depending on how many pools MinSwap expects to exist, this can be a non-negligible amount of Ada. This behavior is not explicit in the specification.

2.2.4 Code Quality

2.2.4.1 Reliance on Indexes Into ScriptContexts' `txInputs` and `txOutputs`

Severity: Medium

Some of the validators rely on specifying an explicit integer index that indicates where certain transaction input is supposed to be located within the list of `txInfoInputs`. For example, the snippet below was taken from the `validateApplyPool` function, which receives `licenseIndex` in a redeemer:

```
licenseInput :: TxInInfo
licenseInput = txInputs !! licenseIndex

txInputs :: [TxInInfo]
!txInputs = txInfoInputs $ scriptContextTxInfo info
```

The problem is that the order of inputs is supposed to be seen as something non-essential. The very type of a transaction relies on *a set of inputs*[2]. As far as we understand, the motivation for relying on the order of transaction inputs is to process orders chronologically. Yet, batchers are free to ignore that restriction, as we could see in issue 2.2.2.1.

We suggest that enforcing the particular ordering in a batch is not that important. Instead, if the orders are considered in whichever ordering they come (which will most likely be lexicographic on the **`TxOutRef`**), it is very difficult to even attempt any front-running attack. A potential attacker can never rely on their transaction being performed first because a batcher might include an order on a lexicographically smaller **`TxOutRef`**. If the attacker is a batcher, they have the freedom to choose which orders to batch together, which might enable them to pick a set of orders in which theirs comes first. However, as it currently stands, the batcher can already do that for arbitrary sets of orders (issue 2.2.2.1). Removing that constraint will inherently strengthen the protocol against front-running orders.

Maybe a lot of the code can be made simpler if the specification of `validateApplyPool` mentioned it processes the relevant inputs in the order given by the **`ScriptContext`**. We could even think of dropping the constraint of needing a one-to-one relationship between `userInputs` and `userOutputs`, possibly making transactions smaller. Issue 2.2.4.3 explores this direction a little further.

2.2.4.2 Duplication of ScriptContext Definition

Severity: Medium

[2] `https://github.com/input-output-hk/plutus/blob/1f31e640e8a258185db01fa899da63f9018c0e85/plutus-ledger-api/src/Plutus/V1/Ledger/Tx.hs#L118`

We understand that using **`Ledger.Context.ScriptContext`** causes scripts to be more expensive to run due to unnecessary deserialization of potentially unused data. Yet, we have a slight concern that the cognitive overhead, maintainability cost and the potential for bugs are not worth that price. In particular, in MinSwap's own encoding of **`Ledger.Context.ScriptContext`** the very important `getContinuingOutputs` function was forgotten, which opened the vulnerability discussed in issue 2.2.1.3.

2.2.4.3 Large Refactoring Opportunities

Severity: Low

There is heavy code duplication in the order validation code, which could be made more reusable and easier to maintain and understand. In particular, the functions:

- `validateDeposit`
- `validateSwapExactOut`
- `validateSwapExactIn`
- `validateOneSideDeposit`
- `validateWithdraw`

exhibit enough common patterns that it could be worthwhile to rephrase those functions, making them return a **`TxOut`** instead of receiving one and returning a boolean. This would increase code reuse and facilitate further refactorings. In fact, we had to perform that very task in order to rebuild MinSwap's off-chain code in `cooked-validators`. Depending on how much time MinSwap is willing to put in improving the code, it could be interesting to study whether these functions could return and manipulate constraints in such a way that `applyPool` would just ensure the transaction outputs satisfy those constraints, regardless of how many outputs are there. We could get rid of the requirement that each order must have a corresponding UTxO which could yield smaller transactions. We do not have an estimate on whether or not this would make the script more expensive to execute, however.

2.2.4.4 Protocol Specification Lacking

Severity: Medium

The documentation and specification of the protocol leaves room for improvement. There is seldom information on what correct processing of orders should look like. Defining a clear specification is important because it establishes the difference between correct and incorrect behavior. For example, the implementation is indicative that applying orders in a pool could be seen as the operation of an *inverse semigroup*, whose elements are batches of orders and they could be inverted in the respective setting. A more formal description of such operations is invaluable for creating meaningful tests and establishing the criteria of what constitutes correct behavior from what is a bug. In this particular instance, there is no ground truth that we can use to check whether the functions defined in **`Minswap.ConstantProductPool.Utils`** are correct.

2.3 Conclusion

This report outlines the 13 concerns that we have gathered while inspecting the design and code of MinSwap, pertaining to the code contained in the files listed in Table 2.1. As stated in Chapter 1, Tweag

does not recommend for nor against the use of any work referenced in this report. Nevertheless, the existence of *high* and *critical* severity concerns is a warning sign.

Appendix A

Appendix

A.1 Tweag's Modifications

We have applied the three patches listed in Listings 1 to 3, which modify some of the files listed in Table 2.1 making them more amenable to testing.

```
diff --git a/src/Minswap/ConstantProductPool/Types.hs b/src/Minswap/ConstantProductPool/Types.hs
index 8fceb19..1e7bf16 100644
--- a/src/Minswap/ConstantProductPool/Types.hs
+++ b/src/Minswap/ConstantProductPool/Types.hs
@@ -34,7 +34,7 @@ data ProfitSharing = ProfitSharing
   { psFeeTo :: Address,
     psFeeToDatumHash :: Maybe DatumHash
   }
-  deriving (Haskell.Show)
+  deriving (Haskell.Show, Haskell.Eq)

 PlutusTx.makeIsDataIndexed ''ProfitSharing [('ProfitSharing, 0)]
 PlutusTx.makeLift ''ProfitSharing
@@ -52,7 +52,7 @@ data PoolDatum = PoolDatum
     pdRootKLast :: Integer,
     pdProfitSharing :: Maybe ProfitSharing
   }
-  deriving (Haskell.Show)
+  deriving (Haskell.Show, Haskell.Eq)

 PlutusTx.makeIsDataIndexed ''PoolDatum [('PoolDatum, 0)]
 PlutusTx.makeLift ''PoolDatum
@@ -78,7 +78,7 @@ data PoolRedeemer
   | WithdrawLiquidityShare
       { wlsFeeToIndex :: Integer
       }
-  deriving (Haskell.Show)
+  deriving (Haskell.Show, Haskell.Eq)

 PlutusTx.makeIsDataIndexed
   ''PoolRedeemer
```

Listing 1: Patch for **`Minswap.ConstantProductPool.Types`**

```
diff --git a/src/Minswap/ConstantProductPool/OnChain.hs b/src/Minswap/ConstantProductPool/OnChain.hs
index 46c826f..0f6e7b6 100644
--- a/src/Minswap/ConstantProductPool/OnChain.hs
+++ b/src/Minswap/ConstantProductPool/OnChain.hs
@@ -21,6 +21,8 @@

 module Minswap.ConstantProductPool.OnChain
   ( poolCode,
+    mkLicenseSymbol,
+    mkOwnerTokenName
   )
 where
```

Listing 2: Patch for **`Minswap.ConstantProductPool.OnChain`**

```
diff --git a/src/Minswap/BatchOrder/Types.hs b/src/Minswap/BatchOrder/Types.hs
index a1c74b9..dd4c5a2 100644
--- a/src/Minswap/BatchOrder/Types.hs
+++ b/src/Minswap/BatchOrder/Types.hs
@@ -48,7 +48,7 @@ data OrderStep
       { osdDesiredCoin :: AssetClass,
         osdMinimumLP :: Integer
       }
-  deriving (Haskell.Show)
+  deriving (Haskell.Show, Haskell.Eq)

 PlutusTx.makeIsDataIndexed
   ''OrderStep
@@ -68,12 +68,13 @@ data OrderDatum = OrderDatum
     odBatcherFee :: Integer,
     odOutputADA :: Integer
   }
-  deriving (Haskell.Show)
+  deriving (Haskell.Show, Haskell.Eq)

 PlutusTx.makeIsDataIndexed ''OrderDatum [('OrderDatum, 0)]
 PlutusTx.makeLift ''OrderDatum

 data OrderRedeemer = ApplyOrder | CancelOrder
+  deriving (Haskell.Eq)

 PlutusTx.makeIsDataIndexed
   ''OrderRedeemer
```

Listing 3: Patch for **`Minswap.BatchOrder.Types`**