MuesliSwap audit report by MLabs

Finished 2022-02-22

Audited code

The code audited is specifically commit `5b33c2906a80c747108fae1e0885d62ffeaee4de`.
We assume that the *behavior* of the output generated by PlutusTx is deterministic, i.e. the compiler is not buggy.

After reporting the initial findings to the MuesliSwap team, the new code with the fixes has been also audited. The review audit is for commit `e35e63a776c968136cc3b5aa814cc7b9dce9b33b`.

The audited file, where all the on-chain logic is contained is `muesliswap-contract-audit/order-validator/src/Cardano/MuesliSwapOrderValidator/OrderValidator.hs`

The hash of the final file is `be5313aa9d6bad72f8703a2b02ac2f9ee3b003b5578f8a9ffb0c9b2b9c09fa5a`.

We used commit `3d1b5f423146eb0e6a6ac35aea0f354c9115f493` of https://github.com/input-output-hk/plutus and commit **46e831e49287c3a04f83497b9dd9e70718092cde** of https://github.com/input-output-hk/plutus-apps.

With this environment, the hash of the final audited script is `0e198d114bb14a9a44ee57428ff114d3c4b12f32a8c5aa8ec2a49db0`

No off-chain code has been audited.

Summary

In general, after the initial audit and corresponding revision, MuesliSwap v2 does not have any critical issues.
The two cases the MuesliSwap developers wanted us to check for were:

- If `oAllowPartial` is **not** set, then the consuming transaction must either be canceling the order (and thus be signed by the creator), or the creator must be receiving the full amount specified.
- If `oAllowPartial` is set, then the owner of the creator either eventually receives the full amount specified, or by canceling gets back *r* times the amount they desired to buy, and *1-r* times the amount they initially set to sell, where 0<r<1. Both quantities should be rounded up to the next integer.

Both were found to be true.

In addition, the script size was only 6316 bytes when we compiled it (6992 bytes after the revision), which is roughly half of what the MuesliSwap developers told us. We suspect this to be because they calculated the size of the hexadecimal formatted serialization, which is roughly twice as big as the raw version.

Methodology

We have inspected the code and made a test suite to test for the corner cases of the code. Through this we have found issues which have been communicated to the developer team and then fixed.

Primarily, we tried to find counterexamples where the assumptions above did not hold.
Such an example is a transaction with more than one script at play, which is where the double satisfaction attack is possible, albeit in the latest version of MuesliSwap v2, this exploit is fixed.
We have also looked for issues that arise from discrepancies between Haskell and Plutus, such as the difference in strictness.

The final version of the test suite is commit `6ecd17ab17cb0817b3dcfd2948c0a3a6ca07cb2f`.

All the described tests are located under `test/Suite/UnitTests`. Tests especially designed to demonstrate the double satisfaction attack are placed under `test/Suite/UnitTests/DoubleSatisfaction` while the rest of them are placed under `test/Suite/UnitTests/EmulatorTrace`.

Disclaimer

This report is for informational purposes only, and does not constitute financial or legal advice. Our auditors have worked diligently to find any and all bugs and vulnerabilities, however this process involves manual inspection and work which is subject to human error. Neither MLabs nor its auditors can assume any liability whatsoever for the use, deployment or operations of the smart contracts. Smart contracts can be invoked by anyone on the public internet and as such carry substantial risk.

Severity classification

The issues described in this report are classified according to their severity in four levels of priority, namely P1 to P4. The description of each category is as follows:

- P1 - Catastrophic issue, which leads to loss of all or most of the user's funds
- P2 - Critical issue, which may lead to loss of a part of the user's funds
- P3 - Non-critical issue which does not result in a loss of funds, but is still important
- P4 - Code quality issue

Found vulnerabilities

A01. Double satisfaction attack

Priority: P1
Status: Fixed

Description

The basic double satisfaction attack is described in https://github.com/input-output-hk/plutus/blob/master/doc/reference/common-weaknesses/double-satisfaction.rst

When used in isolation (i.e., no other scripts are present as inputs) the order validator script is protected against the basic scenario outlined in the link above. However, the protocol is vulnerable to double satisfaction attacks by using a second script.

Tests using a MuesliSwap clone

This attack can be first observed by just using a clone of the MuesliSwap order validator. We have copied the code and changed it slightly so the logic is not changed, but the hash changes. The script used is `src/Cardano/MuesliSwapOrderValidator/Secondary/OrderValidatorClone.hs`, adding the unnecessary check `(oCreator ownIn0) == (oCreator ownIn0)`, which is always true.

The scenario for the attack is outlined in `test/Suite/UnitTests/DoubleSatisfaction/DEX/Clone/Trace.hs` as follows (throughout the trace we use the convention to call "seller" to the user selling native tokens and "buyer" to the user selling ADA):

- Alice (the seller) places an order to sell a Token A and receive X amount of ADA in the original MuesliSwap script.
- Alice places another order to sell a Token B (could be the same as Token A or different) and receive Y amount of ADA in the clone script.
- Bob places an order to buy Token B in the clone script, offering max (X,Y) ADA for it.
- Bob creates a transaction that consumes the three orders, paying max (X,Y) ADA to Alice and getting both Token A and Token B.

From the point of view of the original script, Alice got at least the amount of ADA that was asking for Token A, so it passes validation. From the point of view of the clone script, Alice got at least the amount of ADA that was asking for Token B, and Bob got the Token B, so it passes validation. The attack has been completed successfully.

Note: it is not even necessary that Bob places an order in any of the scripts. The attack can be performed by building a transaction that consumes the two orders from Alice and sending the ADA to Alice directly from a wallet controlled by Bob. This scenario is also demonstrated as another test in the same file.

Test using MuesliSwap v1.1

To demonstrate the attack with an existing script, a variation of the attack using both MuesliSwap v2 and MuesliSwap v1.1 is provided in `test/Suite/UnitTests/DoubleSatisfaction/DEX/V1/Trace.hs`.

The setup is analogous to the above scenario, but this time Token B is sent to the MuesliSwap v1.1 instead of the MuesliSwap v2 clone.

Note that in this case, it is not possible to perform the attack directly from the wallet controlled by Bob. In order to successfully perform the attack, Bob has to first place an order in the v1.1 script, since this version of the protocol checks that the value comes from the script.

Test using a non-DEX script

Lastly, it is worth mentioning that this attack does not require the presence of two DEXs or marketplaces. Any kind of script can be used as the second script as long as it is not protected against this kind of attack.

For instance, let us consider the reward script in `src/Cardano/MuesliSwapOrderValidator/Secondary/Reward.hs`. This simple script locks a UTxO that can be consumed only if the value is spent to the public key specified in its datum.

The attack described in `test/Suite/UnitTests/DoubleSatisfaction/Reward/Trace.hs` goes as follows:

- Alice places an order in MuesliSwap, selling Token A and asking X amount of ADA for it.
- Alice is entitled to some reward, for example as part of a staking protocol. This protocol does not send rewards directly to the users, but instead sends them to the described reward script, attaching the PubKeyHash of Alice as datum. In this case the reward is Y ADA, where Y >= X.
- Bob creates a transaction consuming both inputs, paying Y ADA to Alice and getting the Token A.

From the point of view of the MuesliSwap order validator, Alice got at least the X ADA that was asking for, so it passes validation. From the point of view of the reward script, the value was sent to Alice, so it passes validation. The attack is performed successfully, and in this case the attacker (Bob) did not have to spend any funds at all. Token A was paid with Alice's own funds.

Further notes and limitations

- The tests described above show that the double satisfaction attack can be carried out between MuesliSwap v2 and any other protocol that is vulnerable. This would include a potential MuesliSwap v3. For this to be avoided, **both** scripts must check that the payment output is unique (by e.g. attaching a unique datum to it). If only one does this, then the combination will still be vulnerable.

- The attack can be carried out without spending any funds at all, by using the victim's own funds or simply by matching other user's orders in two different marketplaces.

- The attack is only feasible if multiple scripts can fit into a single transaction. This is currently possible with just MuesliSwap v1.1 and v2, because combined they leave enough space for a few UTXOs and some other metadata in the transaction.

Suggested fix

Attach a unique datum to the payment UTXOs that is e.g. derived from the `TxOutRef`s of the relevant inputs.

Comments on status

This issue has been fixed in commit `2ec7f9bd2ea8aa353dc6ebece375fe68826d9dec`.

The new version of MuesliSwap v2 avoids this problem by attaching a datum to the payment UTXOs (locked with a PKH). The fix should work as long as no other protocol coincidentally decides to use the same datum, i.e. a datum that contains `"MuesliSwap_v2"`.

It is important to note that although no funds can be stolen from MuesliSwap v2 after the fix, it still could be involved in transactions where funds are stolen. Even if MuesliSwap v2 is protected against double satisfaction attack, when interacting with an unprotected script a double satisfaction attack could still succeed. However, in this case the stolen funds would be the ones locked by the other script.

An exception to this would be when interacting with MuesliSwap v1.1. Although the old version of MuesliSwap is not protected against double satisfaction attacks, the new MuesliSwap v2 explicitly verifies that there are no funds locked by MuesliSwap v1.1 being spent in the same transaction.

A02. Inverted logic in `validPartial` allows buying for pennies

Priority: P1
Status: Fixed

Description

This is quite a fatal attack. Consider an `r` that is as small as possible, i.e. morally `1/∞`. In such a case, `justBought` and `justSold` will be tiny, and only a tiny amount will be bought and sold. However, as it is a partial match, there must be exactly one continuing output corresponding to this order. On line 202, it is asserted that the new order is equivalent to the old one, except for `oBuyAmount`. `oBuyAmount` is such that `new = old * r`. This is incorrect, as the resulting ratio of what is to be sold and bought will be different for the new order relative to the original order. The lower `r` is, the lower the amount of `buyToken` to be bought will be in the new order.

Tests showing the described behavior

In `test/Suite/UnitTests/EmulatorTrace/Trace.hs` there are two tests related to this attack.

The test labeled "Unfair asymmetric partial swaps" successfully simulates the following scenario:

- Alice places an order to buy 100 MILK and offers 100 ADA (Order 1).
- Bob places an order to buy 10 ADA and offers 10 MILK (Order 2).
- Bob places another order to buy 10 ADA and offers 10 MILK (Order 3).
- Bob matches Order 1 (partial with r = 1/10) and Order 2 (full). Alice gets 10 MILK (locked in the script) and Bob gets 10 ADA. Order 1 still offers 90 ADA and the amount of MILK it is asking for is updated to 100 * 1/10 = 10 MILK.
- Bob matches Order 1 (full) with Order 3 (full). Alice gets 10 MILK (+ the 10 MILK locked) and Bob gets 90 ADA.

In the end, Alice got 20 MILK instead of the 100 MILK that was asking for in the first place, and paid 100 ADA. Bob got 100 ADA and only had to pay 20 MILK.

On the other hand, the test labeled "Fair asymmetric partial swaps" tries to simulate the fair scenario, but it fails:

- Alice places an order to buy 100 MILK and offers 100 ADA (Order 1).
- Bob places an order to buy 10 ADA and offers 10 MILK (Order 2).
- Bob places another order to buy 90 ADA and offers 90 MILK (Order 3).
- Alice matches Order 1 (partial with r = 1/10) and Order 2 (full). Alice gets 10 MILK (locked in the script) and Bob gets 10 ADA. Order 1 still offers 90 ADA and the amount of MILK it is asking for is updated to 90 MILK. ***TRANSACTION FAILS***

Suggested fix

Invert the inverted logic to fix it.

Comments on status

This issue was correctly fixed in commit `2817804dcf5c6328f51cd1b1ecc2692bfd0d8be9`.

A03. Ratio is skewed towards matcher rather than creator of order on partial matches

Priority: P2
Status: Fixed

Description

The rounding logic is the opposite of what it should be. If the value that the creator of the order receives, `justBought`, has `floor` applied, then a lower output value would succeed to validate. Likewise, if the value that the creator of the order gives, `justSold`, has `ceil` applied, again a lower output value would succeed to validate. This means that the creator of the order could get less value than expected.

Tests showing the described behavior

In `test/Suite/UnitTests/EmulatorTrace/Trace.hs` there are three tests related to this attack.

The test labeled "Extract value through rounding" successfully exploits the fact that `justSold` has `ceil` applied instead of `floor`:

- Bob places an order to sell 3 MILK and asks for 30 ADA. Therefore, the market value (or at least the perceived value) of each MILK token is 10 ADA. (Order 1)
- Alice places an order to sell 15 ADA and asks for 2 MILK (Order 2).
- Alice matches Order 1 (partial, with r = 1/2) with Order 2 (full). Alice gets 2 MILK and Bob gets 15 ADA (locked in the script). Order 1 still offers 1 MILK and the datum is updated to ask for 15 ADA.
- Since now it is asking for too much ADA, nobody wants to match the order anymore. Bob cancels the order and gets the 1 MILK back, along with the locked 15 ADA.

In the end Bob ends up with 1 MILK and 15 ADA, valued at 25 ADA. Bob is not happy.

The fact that `justBought` has `floor` applied instead of `ceil` cannot be exploited, because of the constraint on line 204 which ensures that `r` multiplied by `oBuyAmount ownInO` is always a natural.

The test labeled "Buy amount not divisible, rest ok" tries to match a partial order with an `r` that when multiplied buy `oBuyAmount` does not result in a natural:

- Alice places an order to sell 5 ADA and asks for 5 MILK (Order 1).
- Bob places an order to sell 3 MILK and asks for 2.5 ADA (Order 2).
- Bob matches Order 1 (partial, with r = 1/2) with Order 2 (full). Alice gets 3 MILK (locked in script) and Bob gets 2.5 ADA. Order 1 still has 2.5 ADA left and the amount of MILK it is asking for should be updated to 5 * 1/2 = 5/2 = 2.5, but that is not possible because

the `oBuyAmount` has to be an integer, so it gets rounded to 3 MILK (to protect Alice).
TRANSACTION FAILS

Let us assume that the code was changed to allow for free choice of `r` and corresponding rounding when updating `oBuyAmount`. In this case the fact that `justBought` has `floor` applied instead of `ceil` can now be exploited, as shown by the test labeled "Underpay through rounding":

- Alice places an order to sell 5 ADA and asks for 5 MILK (the market value is 1 ADA = 1 MILK) (Order 1).
- Bob places an order to sell 2 MILK and asks for 2.5 ADA (Order 2).
- Bob matches Order 1 (partial, with `r = 1/2`) with Order 2 (full). Bob gets 2.5 ADA and Alice gets 2 MILK (locked in script). Order 1 still has 2.5 ADA left and asks for 3 MILK.
- Alice cancels the order and gets back 2.5 ADA and 2 MILK, valued at 4.5 ADA. Alice is not happy.

Note: for this test to succeed, the order validator logic has to be changed. It is enough to comment out the check in line 204 regarding the datum update.

Suggested fix

This should be reversed. To ensure that the ratio between what is bought and sold is not **below** what the creator specified, detrimental, what is bought should have `ceil` applied and what is sold should have `floor` applied.

Comments on status

This issue was mostly fixed in commit `2817804dcf5c6328f51cd1b1ecc2692bfd0d8be9`.

What is being sold, `justSold`, is correctly rounded down (using `truncate`). However, `justBought` is still being rounded down (also using `truncate`). Although it is conceptually wrong, it is not a problem in practice, as it is always a natural because of the check that `oBuyAmount ownInO * (1 - r)` equals a natural.

Functionality-wise, it's fine, but it could become a problem if the logic were changed in a future version and the mentioned check were removed.

Code quality issues

B01. Dangerous use of `PlutusTx.unstableMakeIsData`

Priority: P3
Status: Fixed

Description

Use of `PlutusTx.unstableMakeIsData` might lead to unexpected changes of the hash when compiling

Suggested fix

Use `PlutusTx.makeIsDataIndexed` instead.

Comments on status

This issue was correctly fixed in commit `2817804dcf5c6328f51cd1b1ecc2692bfd0d8be9`.

B02. Redundant use of conditional expression

Priority: P4
Status: Fixed

Description

The conditional expression `if x then true else false` expression is redundant, as it can be simplified into just `x` without changing execution semantics.

Suggested fix

Simplify `if x then true else false` into just `x`.

Comments on status

This issue was correctly fixed in commit `2817804dcf5c6328f51cd1b1ecc2692bfd0d8be9`.

B03. Reusable functions in local scope

Priority: P4
Status: Fixed

Description

The functions `ceil` and `floor` implement logic with a scope that goes beyond the **mulFloor** and **mulCeil** functions inside of which are defined, and could be easily reused.

Suggested fix

Both `ceil` and `floor` should be separate functions rather than being tied to multiplication.

Comments on status

This issue was fixed in commit `2817804dcf5c6328f51cd1b1ecc2692bfd0d8be9`, with a caveat. Although `floor` is correctly implemented as a separate function, `ceil` is not implemented anymore. This is because in practice there is no need to round up a rational number throughout the contract. However, it is important to keep in mind that should the logic be updated to allow for a free choice of `r`, the ceil function should be re-implemented (see A03).

B04. Misleading `Eq` instance

Priority: P4
Status: Fixed

Description

The `Eq` instance for `Order` is misleading, as it does not include `oBuyAmount`.

Suggested fix

Define `Eq` for `Order` in a way that all the fields are compared for equality, including `oBuyAmount`.

Comments on status

This issue was fixed in commit `2817804dcf5c6328f51cd1b1ecc2692bfd0d8be9`.

B05. Confusing implementation of rounding function

Priority: P4
Status: Fixed

Description

The function `floor` is not implemented in a clear way. It unnecessarily defines two cases depending on the sign of the inputs, which are always going to be positive. Fortunately, in practice it is not buggy and does not cause problems, as `s` will never be under 0 because the input is always positive (a positive rational is enforced and asking for a negative amount of tokens would not make sense), and `n` will always be returned.

Suggested fix

Reimplement the function `floor` so it is clear that it always returns `n`.

Comments on status

This issue was fixed in commit `2817804dcf5c6328f51cd1b1ecc2692bfd0d8be9`.