Technical Review of Optim

High Assurance Software Team
September 2, 2022

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 Optim'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 Optim. 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 analysed the validator scripts comprising the Optim bonds & pools protocol as of commit `eb6157` … `c7e1d1` of the code repository[1] shared with us. The names of the files considered in this audit and their `sha256sum` are listed in Table 2.1.

`sha256sum`	File Name
`1938e8` … `863b8f`	`offchain/data/compiled-scripts/pool-token-mint-policy.plutus`
`e86b99` … `f5e496`	`offchain/data/compiled-scripts/closed-validator.plutus`
`8a457d` … `fb324e`	`offchain/data/compiled-scripts/open-pool-validator.plutus`
`bef7ff` … `a60b8d`	`offchain/data/compiled-scripts/closed-pool-validator.plutus`
`8f3929` … `e02263`	`offchain/data/compiled-scripts/nft-mint-policy.plutus (unpatched)`
`ea369b` … `01d998`	`offchain/data/compiled-scripts/nft-mint-policy.plutus (patched)`
`a53632` … `208548`	`offchain/data/compiled-scripts/bond-token-mint-policy.plutus`
`a57aa2` … `486f7b`	`offchain/data/compiled-scripts/open-validator.plutus`
`1c040d` … `78834f`	`offchain/data/compiled-scripts/bond-writer-validator.plutus`

Table 2.1: *Scripts analysed, and their `sha256sum`*

Our analysis is based on the documentation provided by Optim as of commit `181e3b` … `16d930` of the spec repository[2] describing different aspects of their design, as well as on Slack conversations with Optim and MLabs. Table 2.2 and its content will be referred to as *the specification* of the protocol.

`sha256sum`	File Name
`273f70` … `1e4f3f`	`spo-bond-pools.md`
`d34b00` … `c8fb1d`	`bonds-deps.svg`
`1e9903` … `bcd6cf`	`pools-deps.svg`
`8472bb` … `fbad06`	`liquidity-bonds.md`

Table 2.2: *Documentation files and their `sha256sum`*

Important disclaimer: During this audit process, a discrepancy between `cardano-ledger` and the emulator from `plutus-apps`, on which `cooked-validators` depends, was found, described in Section A.1. Following that finding, we implemented an emergency patch, which was validated by MLabs. The patch

[1] `https://github.com/mlabs-haskell/optim-onchain`
[2] `https://github.com/mlabs-haskell/optim-spec`

is shown in Section A.2. This explains why Table 2.1 provides two hashes for the same script. Notably, the patch is by no means related to any vulnerability, nor minor finding, in Optim bonds & pools.

2.2 Summary of our findings

Most of the findings illustrated in this report come from a few underlying issues:

a) Very few checks are done when paying to the **`BondWriter`** validator. Indeed, the minting policy of the NFTs cannot know about the **`BondWriter`** validator address since the latter is already built from the hash of the former, resulting in a circular dependency.

b) The `uniqNFT`, which supposedly should always be locked by one of the product's scripts, can in fact be redirected to the outside world, leading to many abnormal sequences of actions.

c) Similarly, no checks are done while pools are created. In addition, a pool can match any bond that satisfies its datum's requirements, even if the number of pool and bond tokens do not match.

d) The **`ClosedPool`** validator allows for tampering with the datum when redeeming pool tokens.

Severity	Section	Summary
Critical	2.3.1.8	Pools can be emptied by matching a small bond
Critical	2.3.1.9	Pools can be emptied by matching a bond with the cancel redeemer
Critical	2.3.1.10	Funds can be locked after datum tampering
Critical	2.3.1.11	Bond tokens can be stolen after datum tampering
Critical	2.3.1.12	Bond tokens can be redeemed with different pool tokens
High	2.3.1.1	`uniqNFT` can be redirected when posting a bond
High	2.3.1.2	`uniqNFT` can be redirected when cancelling a bond
High	2.3.1.4	**`Open`** validator can contain unsound bonds
High	2.3.1.5	Funds can be locked with irrevocable staking rights using unsound bonds
High	2.3.1.6	Pool tokens target is unchecked
High	2.3.1.7	Pool target can be compromised using another `uniqNFT`
Medium	2.3.1.3	`uniqNFT` is sometimes required as fees
Medium	2.3.3.2	Initial margin is unchecked
Medium	2.3.3.3	Bond tokens and pool tokens are handled differently
Low	2.3.2.1	Off-chain: Fees are not used in `mkOpenDatum`
Low	2.3.3.1	Buffer boundary is open on the right
Lowest	2.3.2.2	Off-chain: Computation time is wrong

Table 2.3: *Detailed findings*

These high-level findings correspond to a set of low-level findings. More precisely, Table 2.3 lists our concerns with Optim's current code, based on our *partial exploration* during a *limited period of time*, therefore, *Table 2.3 is not guaranteed to be exhaustive*. Throughout the next section, we detail each of our findings. Most of them correspond to issues in the on-chain files that were audited, however, a

rudimentary look was taken as well at the off-chain code provided by Optim. Some minor bugs (*Low* or *Lowest* severity) have been found there which we report as well. It will be made very clear that these correspond to off-chain issues that are only present for information.

2.3 Detailed findings

2.3.1 Vulnerabilities

2.3.1.1 `uniqNFT` can be redirected when posting a bond

Severity: High

When a user posts a bond, the `uniqNFT` and `ownNFT` are minted. The transaction sends the `ownNFT` to the user and the `uniqNFT` to the **`BondWriter`** validator. However, no check ensures that the transaction indeed sends the `uniqNFT` to the validator, and a user can redirect it to any address of their choice, which leads to some unexpected outcomes (see issues 2.3.1.4 and 2.3.1.5).

⇒ Trace `postBondStealUniqNft` illustrates this issue.

Ideally, this check would be made in the NFT minting policy, which would check that the `uniqNFT` is indeed sent to the **`BondWriter`** validator when posting. However, due to the circular dependency issue mentioned in Section 2.2, this is not possible. Instead, we suggest a two-step process for posting bonds, as outlined in Section 2.4.1.

2.3.1.2 `uniqNFT` can be redirected when cancelling a bond

Severity: High

Not only can the `uniqNFT` be redirected when posting a bond as mentioned above, but it can also be be paid to anyone when cancelling a bond. In that case, the `uniqNFT` should be burned, but the **`BondWriter`** validator makes no such check. This leads to the same unexpected results as before (issues 2.3.1.4 and 2.3.1.5), with the addition of issue 2.3.1.7.

⇒ Trace `cancelBondStealUniqNft` illustrates this issue.

We advise to implement the fix outlined in Section 2.4.2.

2.3.1.3 `uniqNFT` is sometimes required as fees

Severity: Medium

On lines 146-150 of the **`Open`** validator, the Optim fee for a bond is calculated as a percentage of a value containing the `uniqNFT`. When the percentage is strictly less than 100%, this ignores the presence of this `uniqNFT` due to integer division, but this is not the case for 100% Optim fees. In that case, the `uniqNFT` is required as an output both to Optim's address and to the **`Closed`** validator. Such constraints cannot be satisfied since there can be, by nature, only one `uniqNFT`. Hence, such a transaction will not validate, resulting in all funds from the bond to be locked forever.

⇒ Trace `moneyLockedOptimFee` illustrates this issue.

We advise to disallow entirely a 100% Optim fee (a corner case that should not, in practise, happen), by changing line 154 in the **`BondWriter`** validator as follows:

from `3_00 #<= otmFee #&&`

to `3_00 #<= otmFee #&& otmFee #< 100_00 #&&`

2.3.1.4 `Open` validator can contain unsound bonds

Severity: High

As mentioned in issues 2.3.1.1 and 2.3.1.2, a user can redirect the `uniqNFT` to a public key address when posting or cancelling a bond. The user can then pay that `uniqNFT` to the **`Open`** validator alongside a suitable datum, giving the impression that the resulting UTxO represents an actual written bond, where anyone can add margin. For any such added margin, the user gets staking rights over those funds.

⇒ Traces `postFakeOpenBond` and `postFakeOpenBondCancel` illustrate this issue.

We advise to implement the fixes outlined in Sections 2.4.1 and 2.4.2 to avoid this issue.

2.3.1.5 Funds can be locked with irrevocable staking rights using unsound bonds

Severity: High

This issue is an extension of the previous one. Here, however, we emphasize that any margin added to the **`Open`** validator by any user will not be redeemable after the bond is closed. The funds will then be locked forever as there are no bond tokens present to exchange, while the attacker retains irrevocable staking rights over any added margin.

⇒ Traces `moneyLocked` and `moneyLockedCancel` illustrate this issue.

As before, we advise to implement the fixes outlined in Sections 2.4.1 and 2.4.2.

2.3.1.6 Pool tokens target is unchecked

Severity: High

When a user creates a pool, the pool tokens (besides the ones bought by that user) are sent to the **`OpenPool`** script. However, as there are no checks made on the expected target of the pool tokens, a user can create a pool, not buy any pool token, but redirect an arbitrary number of them to their own address without any cost. Additionally, any user can actually just mint pool tokens without creating a pool to a similar end.

⇒ Traces `createPoolStealPoolTokens` and `mintPoolTokensWithoutPool` illustrate this issue.

The first part of this vulnerability is known to Optim; and it was presented to us as a design decision, since it was assumed stealing pool tokens could not yield any benefit for an attacker. However, as we show in issue 2.3.1.12, this can allow an attacker to steal bond tokens. We advise to implement the two-step fix outlined in Section 2.4.1 to fix this issue.

2.3.1.7 Pool target can be compromised using another `uniqNFT`

Severity: High

Redirecting the `uniqNFT` during cancellation can also cause repercussions in relation to pools, by targeting a different bond than expected. Specifically, a user A can post an enticing bond b_1, so a different user B creates a pool targeting b_1. Then, A can cancel b_1 while retrieving the associated `uniqNFT`. Next, A can post a second, less enticing bond b_2 while paying the `uniqNFT` of b_1 as part of the initial margin. Finally, provided all the pool tokens have been bought, A can match the targeted pool to b_2, despite that bond not being the bond that B was targeting, since b_2 contains the `uniqNFT` of b_1. While the parameters of b_2 still must satisfy the requirements of the created pool's datum, this can prevent buyers of the pool tokens to get the expected rewards. Indeed, since B targeted a specific bond, they have no incentive to provide matching parameters in the datum that would differ from the default ones.

⇒ Trace `targetWrongBondWithExtraUniqNft` illustrates this issue.

As this issue specifically arises from the redirection of the `uniqNFT` during cancellation, we suggest implementing the fix outlined in Section 2.4.2.

2.3.1.8 Pools can be emptied by matching a small bond

Severity: Critical

An invariant in the pools is that the number of bond tokens should equal the number of pool tokens. However, in the **`OpenPool`** validator, there is no check regarding this equality, which results in a critical vulnerability when a bond is created with a very small number of bond tokens. Once a pool is created and all pool tokens have been bought, an attacker can post a bond with only one bond token, and match the pool with that bond. Thus, it becomes possible to redirect the ADA value of all the pool tokens (minus one) to an attacker's public key address, essentially emptying the pool. In addition, it is possible to perform this attack with a bond written by a different peer, provided the number of bond tokens in that bond is less than the intended pool size.

⇒ Traces `redirectExtraOwnBond` and `redirectExtraOtherBond` illustrate this issue.

To fix this, we advise to check that the number of bond tokens is equal to the pool size, as described in Section 2.4.4. Notably, it was mentioned during a meeting with MLabs that this issue was also found by them and fixed in later commits than the one we analysed.

2.3.1.9 Pools can be emptied by matching a bond with the cancel redeemer

Severity: Critical

While the previous issue allows an attacker to steal an ADA amount equivalent to all but one of the pool tokens, it is possible to steal them all. Indeed, when matching a bond with a pool, the **`OpenPool`** validator assumes that the input from the **`BondWriter`** validator is redeemed using the **`WriteBond`** redeemer, but no such check is done. Any attacker can change that redeemer to **`CancelBond`** and redirect all the funds contained in the pool to any address, since cancelling a bond does not require any output to the **`Open`** validator.

⇒ Trace `matchWithWrongRedeemer` illustrates this issue.

Fortunately, this can only be done when matching an artificial bond with 0 as its `bondamount`, since any other amount would trigger a check that some bond tokens are paid to the **`Open`** validator. Thus, we suggest to implement the fix in Section 2.4.4 which should be sufficient. However, we advise to be very careful whenever a product consumes a UTxO from another product and to make sure that the consumption is done using the right redeemer to trigger the adequate checks.

2.3.1.10 ■ Funds can be locked after datum tampering

Severity: Critical

When a user redeems some (but not all) pool tokens for bond tokens from the **`ClosedPool`** validator, the user must pay back all bond tokens not withdrawn. The datum paid back to the **`ClosedPool`** validator should be identical to the one spent, but this is not checked. As a result, an attacker can redeem some pool tokens for bond tokens while tampering with either the bond symbol or the bond token within the **`ClosedPool`** validator's datum, such that no other partial redemptions are possible. Redemption of all remaining tokens would still be possible, but this would hardly happen since pool tokens are meant to be split into many buyers, each of which could not afford to match the whole bond.

⇒ Traces `tamperBondSymbolPartialWithdrawal` and `tamperBondTokenPartialWithdrawal` illustrate this issue.

To address this, we advise to prevent such datum tampering as depicted in Section 2.4.3.

2.3.1.11 ■ Bond tokens can be stolen after datum tampering

Severity: Critical

The above issue can be extended to another vulnerability, where all the bond tokens from a bond can be stolen. This is done by tampering with the bond token in the **`ClosedPool`** datum, while using an additional bond in order to pass the validator checks, and in turn emptying all the bond tokens from the pool.

This begins by an attacker posting and writing a bond b_1 with two bond tokens. Then, the attacker must buy two pool tokens from a pool p. Once p is fully funded, the attacker matches p to a different bond b_2 (which may or may not have been posted by the attacker) while sending the bond tokens of b_1 to the **`ClosedPool`** validator. The attacker next redeems one pool token of p for a bond token of b_2 from the **`ClosedPool`** validator, while tampering with the **`ClosedPool`** datum, changing the bond token name in the datum from b_2 to b_1. Finally, the attacker can redeem the other pool token for the bond token of b_1 in the **`ClosedPool`** validator, which will successfully validate because the datum has changed. In doing so, the attacker empties all of the bond tokens of b_2. At that point, the attacker can redeem those bond tokens for ADA once b_2 expires.

⇒ Trace `poolWithOtherBondToken` illustrates this issue.

As above, the simplest fix for this attack would be to disallow tampering with the datum in the **`ClosedPool`** validator, outlined in Section 2.4.3.

2.3.1.12 ■ Bond tokens can be redeemed with different pool tokens

Severity: Critical

As mentioned in issue 2.3.1.6, an attacker can create a pool, and redirect the pool tokens to their public

key address instead of paying them to a script. After doing this, an attacker can buy exactly one pool token from a different pool. Then, the attacker can redeem that pool token while tampering with the **`ClosedPool`** datum, changing the pool token in the datum to the pool token of the pool the user originally created. Finally, the attacker can use their original pool tokens, which are not matched to any bond, to redeem the bond tokens from the honestly created pool, thereby emptying a pool for the price of one pool token.

For this issue, it is worth noting that one can use any pool token to redeem any bond token, provided they have at least one copy of the correct pool token in order to tamper the datum. That is, the pool tokens do not need to be stolen as in issue 2.3.1.6 in order for a user to perform this attack.

⇒ Traces `poolWithOtherFakePoolToken` and `poolWithOtherRealPoolToken` illustrate this issue.

We advise to implement both the two-step process for creating a pool as outlined in Section 2.4.1, as well as the datum tampering fix outlined in Section 2.4.3 in order to solve this issue.

2.3.2 Implementation Bugs

2.3.2.1 ■ Off-chain: Fees are not used in `mkOpenDatum`

Severity: Low

The off-chain function `mkOpenDatum` takes as a parameter the **`BondWriter`** datum, which includes the Optim fee as a field. However, `mkOpenDatum` will always set the Optim fee in the resulting **`Open`** datum to be **`BasisPoints`** `300` (or 3%), instead of retrieving the Optim fee from the **`BondWriter`** datum. We advise to change line 84 of `Test/Optim/Bonds/App/Bonds.hs`

from
```
openDatum'otmFee = BasisPoints 300,
```
to
```
openDatum'otmFee = bondWriterDatum'otmFee,
```

2.3.2.2 ■ Off-chain: Computation time is wrong

Severity: Lowest

The off-chain function `fromEpochTime` appearing in the file `Test/Optim/Common.hs` is missing the addition to the origin of time when computing the current slot. We advise to change the corresponding expression in `fromEpochTime`

from
```
POSIXTime $ n * getPOSIXTime epochLength
```
to
```
POSIXTime $ n * getPOSIXTime epochLength + epochBoundary
```

2.3.3 Unclear Specification

2.3.3.1 ■ Buffer boundary is open on the right

Severity: Low

In line 156 of the **`BondWriter`** validator, buffer is checked to be in $[0, \mathit{duration})$ and it is unclear why the right boundary is not included in the interval. In other words, why not $[0, \mathit{duration}]$? In that case the whole margin should be added in a single original step when the buffer actually equals the duration.

2.3.3.2 Initial margin is unchecked

Severity: Medium

When adding margin to a bond that was already created, a user is only permitted to add margin exactly corresponding to the ADA value for some number of epochs. If a user attempts to add, for example, 1.5 epochs of margin, this transaction will not validate. However, when creating a bond, a user can include *any* value as prepaid margin. This includes ADA values not precisely corresponding to an epoch, or even non-ADA values. This causes some peculiar behaviour, most notably in issue 2.3.1.7, where an additional `uniqNFT` is added as prepaid margin, leading to a vulnerability.

To fix this, we advise to implement the two-step process for posting a bond a depicted in Section 2.4.1 which would allow the script to make checks over the initial value. These checks could be similar to the `onMarginAdd` case in the **`Open`** validator (notably the `correctAmount` check on line 113), or at the very least, could disallow non-ADA values as prepaid margin when posting a bond.

2.3.3.3 Bond tokens and pool tokens are handled differently

Severity: Medium

Bond tokens and pool tokens are essentially two faces of the same coin, as they exhibit similar behaviour. They are all minted at once, they can be bought on a secondary market and eventually will be redeemed either for bond tokens in the case of pool tokens, or for ADA in the case of bond tokens. However, their burning especially is handled in a different manner, since pool tokens are all burned at once while bond tokens are burned whenever a user redeems them for ADA. This discrepancy, which is not the origin of any vulnerability, is a red flag for us because it shows a break of consistency in the application. Such a disparity leads to different checks depending on the nature of the tokens while a single well-defined approach to handle those would lead to less room for error.

2.3.4 Code Quality

We have no concern about code quality.

2.4 Possible solutions

Here is a list of fixes that could help in solving the issues depicted in Table 2.3. Note that we offer *no guarantee* that these would make the product foolproof nor introduce new weak points, which we are not able to assess without a re-audit. So, the following are merely *suggestions* that we believe could help improve the overall robustness of Optim bonds & pools.

2.4.1 Implement a two-step bond and pool posting

Cardano blockchain and associated scripts suffer from a well-known circular dependency issue, where a minting policy for instance would need to have a static parameter corresponding to the address of a script, while said script would need the hash of said minting policy as a static parameter as well. This, of course, is impossible. A possible fix would be to allow multi-purpose scripts as discussed with MLabs, but this is a design choice that is not up to them, or us, to make.

There are, however, ways around this circular dependency issue, one of which we advise to implement when posting a bond and when creating a pool.

The idea is to turn any initial payment to a script which induces a minting of tokens into a two-transactions process. The first transaction does not involve any checks at all, does not mint any tokens that should be locked in the script, and creates "unchecked" UTxOs at the recipient script. The second transaction consumes unchecked UTxOs (with an additional **`Check`** redeemer), mints the required tokens, and pays a "checked" UTxO back to the same script, which contains the newly minted tokens as a proof of their soundness. Since the second transaction uses a redeemer, it can make whatever checks are needed to ensure the tokens are minted correctly and paid to the correct script.

This should solve the issue because the validator itself knows its own address and can thus ensure that the minted tokens are given to itself and cannot be redirected to any random public key by an attacker. Any further transaction consuming one such UTxO would have to rely on the presence of the minted tokens to assess its correctness.

2.4.2 Ensure `uniqNFT` is burned when cancelling

For this issue, it is easy to add a check that the **`BondWriter`** validator burns the `uniqNFT` when cancelling a bond:

```
uniqNftBurned = -1 #== pvalueOf # infoFs.mint # uniqNft # datumFs.tokenName
```

Note that this is identical to a check already present in the **`Closed`** validator. Along with the above fix with the two-step bond posting, this should fix the issues in 2.3.1.2, 2.3.1.4, 2.3.1.5, and 2.3.1.7.

2.4.3 Prevent datum tampering

In almost every relevant transaction from Optim bonds & pools, any output datum is checked to be correct according to the input datum, provided there is one. Often, many fields in the datum are unchanged from input to output, and the validators will ensure this. However, in the **`ClosedPool`** validator, no such check is made, allowing a user to tamper with the **`ClosedPool`** datum when partially redeeming pool tokens. Not explicitly checking this leads to some critical vulnerabilities as seen in issues 2.3.1.10, 2.3.1.11, and 2.3.1.12. To fix this, we advise to rely on similar checks done in other validators; that is, compute an expected datum from the inputs and compare it for equality with the output datum.

2.4.4 Match pool size and bond amount

For this issue, we advise to add the pool size as a parameter to the **`OpenPool`** validator and to check that the number of bond tokens minted is equal to the pool size. This ensures that matching a bond with a small amount of bond tokens will fail, which disallows the critical vulnerabilities present in issues 2.3.1.8 and 2.3.1.9.

As we mentioned in issue 2.3.1.8, MLabs did find this issue in parallel to this audit, and implemented a different fix.

2.4.5 Additional Solutions

The above solutions all fix more than one of the issues from Table 2.3. For additional solutions that only fix one specific issue, refer to the description of the issue in question, namely issues 2.3.1.3, 2.3.2.1, and 2.3.2.2.

2.5 Conclusion

This report outlines the 17 concerns that we have gathered while inspecting the design and code of Optim, pertaining to the code contained in the files listed in Table 2.1. This is *not an exhaustive list* of issues but only those that we have identified during the *limited* time that the audit was conducted. 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 Discrepancy between `cardano-ledger` and `plutus-apps`

The discrepancy happens when relying on the presence of a 0-ADA entry in any minted value, which is always present when using the actual ledger and absent when using the emulator from `plutus-apps`. As an example, the following code:

```
validate _ _ scriptCtx = case Map.lookup (txInfoMint $ txInfo scriptCtx) adaSymbol of
  [_] -> True
  _ -> False
```

is equivalent to `const True` in the `cardano-ledger` and `const False` on the emulator. This led to an issue report for `plutus-apps` (see `https://github.com/input-output-hk/plutus-apps/issues/604`) and the deployment of an emergency patch for the current audit described in Section A.2.

A.2 Patch applied on `Nft.hs`

Figure A.1 depicts our patch to solve the discrepancy between `cardano-ledger` and `plutus-apps`. This patch is done on `Nft.hs` which is the source code file from which `nft-mint-policy.plutus` is generated.

```
diff --git a/onchain/optim-bonds/src/Optim/Onchain/Bonds/Plutarch/Contracts/Nft.hs b/onchain/optim-bonds/src/Optim/Onchain/Bonds/Plutarch/Contracts/Nft.hs
index b4e8088..1dc0fb3 100644
--- a/onchain/optim-bonds/src/Optim/Onchain/Bonds/Plutarch/Contracts/Nft.hs
+++ b/onchain/optim-bonds/src/Optim/Onchain/Bonds/Plutarch/Contracts/Nft.hs
@@ -3,15 +3,14 @@ module Optim.Onchain.Bonds.Plutarch.Contracts.Nft (
   mkNftMintPolicy,
 ) where

+import Prelude (($), (.))
+import Prelude (($), (.), (+), (-))
 import Plutarch.Prelude
 import Plutarch.Monadic qualified as P
 import Plutarch.Api.V1 (
-  PCurrencySymbol,
   PMintingPolicy,
-  PScriptPurpose (PMinting),
-  PTokenName,
   PScriptPurpose (PMinting)
 )
+import Plutarch.Api.V1.Value as Value
 import Optim.Onchain.Common.Plutarch (
   check,
   fromTypedMintingPolicy,
@@ -37,6 +36,12 @@ mkNftMintPolicyTerm =
     let currSym = pfield @"_0" # currSymFs

     mint <- plet txInfoFs.mint
+
+    -- Artificially adding a zero ada entry in the minted Value
+    let oneAda = Value.psingleton # padaSymbol # padaToken # 1
+        zeroAda = Value.punionWith # plam (-) # oneAda # oneAda
+        newMint = Value.punionWith # plam (+) # mint # zeroAda
+
     assets <- plet . pto $ plookupCurrencySymbol # mint # currSym

     pmatch redeemer $ \case
@@ -49,7 +54,7 @@ mkNftMintPolicyTerm =
                                  # assets
         assetValues <- plet $ pmap # plam (\asset -> pfromData $ psndBuiltin # asset)
                                  # assets
-        precNftAssets <- plet . pto $ plookupCurrencySymbol # mint # precNftCs
+        precNftAssets <- plet . pto $ plookupCurrencySymbol # newMint # precNftCs

         ptraceIfFalse "Ref UTxO not spent" -- actually won't trace since `pisSubsequenceOf` will error
           (pisSubsequenceOf # refs # inputRefs) #&&
```

FIGURE A.1: *Patch applied on* `Nft.hs`