TWEAG

Technical Review of Meld Smart Contract

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 MELD'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 MELD. 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

MELD, referred in the following as the client, submitted three Plutus smart contracts to Tweag for auditing: namely the *Vesting* contract, the *Locked Staking* contract, and the *Variable Staking* contract. The client could not provide the *Variable Staking* contract in time for Tweag to work on it within the time allocated to the audit.

Tweag analyzed the *Vesting* and the *Locked* variant *Staking* on-chain validator scripts as of commit `539df59` of the client's repository at `https://github.com/MELD-labs/smart-contracts`. The relevant filenames, including documentation, and their `sha256sum` are listed in table 2.1.

Section 2.2.3 of the audit goes beyond the stated scope and provide some comments on the *Variable* variant of the *Staking* contract. Due to time constraints, this contract is considered as out of the scope of the audit, and the remarks on it are provided out of courtesy to the client.

2.1.1 Auditing process

For each contract, Tweag carried out the following process:

a) Analysis of the official specifications as stated in the documentation provided in `vesting/README.md` and `staking/README.md` and the associated on-chain code in `.../OnChain.hs`, `.../Types.hs`, and `.../Common.hs` files.

b) Informal discussion with the client about general features of the contracts, specific points of context, and technical aspects Tweag wanted to clarify.

c) Simulation of traces in the context of a regular usage of the contracts in order to assert the conformance to the specifications. Unit tests and property-based checking have been carried out and are reported in Section 2.2.1 and Section 2.2.2.1.

d) Identification, from specifications and code inspection, of possible vectors of attack Tweag deemed relevant to explore and experiment on.

e) Attempts to exploit these hypothetical weak points using custom made traces. These attempts are listed as unit tests reported in Section 2.2.2.2 and Section 2.2.2.3.

Throughout this process, Tweag identified side remarks that may also be of interest to the client. They are reported in section 2.2.2.4.

2.1.2 Technical aspects

On the technical level, Tweag carried out its experiments using Tweag's *Cooked Validators* library. It provides tooling for building and manipulating transaction traces. It also interfaces with the Plutus library to validate them.

sha256sum	File Name
c709108...f496b78	vesting/README.md
9af12f7...670baef	vesting/src/MELD/Contracts/Vesting/Common.hs
b46745c...f1abcd1	vesting/src/MELD/Contracts/Vesting/Types.hs
4135a9c...c8a2438	vesting/src/MELD/Contracts/Vesting/OnChain.hs
5b5c5ac...90734a7	staking/README.md
ef4758e...d445e60	staking/src/MELD/Contracts/Staking/Locked/Common.hs
c29825e...bf9855d	staking/src/MELD/Contracts/Staking/Locked/Types.hs
80ac560...dd9c3e2	staking/src/MELD/Contracts/Staking/Locked/OnChain.hs
2cf6d62...e288bdd	staking/src/MELD/Contracts/Staking/Locked/Treasury/Types.hs
ebc0347...cd3ca56	staking/src/MELD/Contracts/Staking/Locked/Treasury/OnChain.hs

TABLE 2.1: *Files containing on-chain code (or documentation) and their* `sha256sum`

Both the *Vesting* and *Staking* contracts mainly deal with a class of assets called *MELD tokens* whose life cycle is outside the scope of the audited smart contracts. Tweag modeled the minting policy of these tokens with a *one shot currency*: a fixed amount of MELD tokens are minted once and for all at the beginning of the test traces and shared among addresses to fit the scenario of each test case.

The *Locked Staking* contract deals with a concept of upgradable non fungible tokens. A token *evolves* when its level (information carried by the token name) increases, which means in practice replacing an NFT by another with the same token name apart from the level. The minting policy for these tokens is not provided by the client alongside the contract. During the audit, Tweag assumed the existence of a minting policy that makes sure minting a higher level NFT requires to burn the associated previous level NFT. Such a policy is paramount to the security of the contract. Section 2.2.2.3 details experiments carried out regarding NFT evolution in the context of a "Locked Staking" position.

2.2 Findings and Deliverables

This section details the performed tests and experiments along with the observations gathered throughout the audit of the client's on-chain validators.

With respect to the functional correctness of the three contracts, we have not found anything out of the ordinary. The validators behave as expected and we did not detect any vulnerability. Our main findings of interest are summed up in table 2.2. They cover a few unexpected behaviours on fringe cases that pose no threat, and some comments on the validators. Testing for the regular usage and failed attempts to violate the specifications are reported in this section but not in the table of findings.

The audit has been carried out using Tweag's *Cooked Validators* library. We provide the client with the audit project alongside this report. We refer to modules of the audit project (`Audit.*`) in the remainder of this report.

2.2.1 Vesting Contract

Severity	Section	Summary
Medium	2.2.1.2	The whole contract relies on one private key
Medium	2.2.2.4	Conditions on the validator check more than what is declared
Low	2.2.2.3	Undeclared NFT in staking position
Low	2.2.2.3	Out of bounds levels
Low	2.2.2.4	Possibility to unstake two positions associated to different authentication asset classes in the same transaction
Low	2.2.2.4	The requirement on the useless treasury inputs makes it harder to collect remainders
Low	2.2.3.2	The Computation of the interest amount is complex and undocumented
Lowest	2.2.2.4	Nested conditions in the validator scripts would benefit from more tracing

TABLE 2.2: *Table of findings*

2.2.1.1 Basic Usage

This contract is very simple, hence allowing almost no attack strategies. We ran some simple traces, all defined in module `Audit.Vesting`. Those traces are really close to the expected usage of the contract.

Possibility to withdraw
The main purpose of the vesting contract is to allow users to withdraw the money stored in their position, with some time constraints.

Hence, the function `withdrawProp` checks that the amount a user can withdraw at each point in time conforms to the specifications of the contract.

Identity of the withdrawer
There are two notions of "withdrawer": it either refers to the person who signs the transaction to withdraw money from a vesting position, or the person who receives the money originating from such a transaction.

Since the specification does not lift the ambiguity, we assumed that signing the transaction was the determining element. Hence, vesting offers the possibility to vest and transfer the money to someone else in the same transaction.

Function `differentSignerReveiverProp` tests this assumption. It turns out our assumption was correct: it is possible to give the money to anyone provided that the signer is the one specified in the datum. Conversely, it is impossible to vest without the signature, even if the money is transferred to its legitimate owner.

Possibility to alter a position
Partial vesting should not make it possible to modify the datum of the vesting position. Especially, it should not make it possible to modify the public key of the expected receiver. Neither should it make it possible to modify the creation date, since it would be a mean to vest without waiting for the contractual epoch. Function `datumVestModifProp` checks that these fields in the datum cannot be changed.

2.2.1.2 On the Administrator

The contract features the `Update` redeemer which allows altering any vesting position on the sole condition that the transaction containing the alteration is signed by the administrator of the contract. Given the freedom this administrator has, it is obvious that if we do not assume any properties on their behaviour, undesired situations may arise. Therefore, we decided not to illustrate those situations. However, this raises a strong concern regarding the administrator position.

The whole contract relies on one private key *Severity: Medium*

The creation and modification of all the vesting positions rely on one single private key. The management of such a key is a major concern for the viability of the contract.

Indeed, if there exists only one copy of the administration key, then the risk of loss is non-negligible. And the loss of this key makes the contract impossible to manage further. On the other hand, having multiple copies of this administration key raises the risk of a malicious use of it, allowing the robber of the key to steal all the meld tokens in every vesting position.

2.2.2 Locked Staking contract

Even if this contract remains quite simple, it is more advanced than *vesting*. Its main purpose is to allow users to stake some Meld tokens for a predefined duration, and to get interest on it at the end of the staking time.

2.2.2.1 Basic usage

The most basic usage of the contract is to simply stake some Melds and wait for when the deadline is reached to unstake them and get the interests. Those basic use cases are explored by functions `traceStake` and `(traceUnstake 1)` of module `Audit.Staking.Locked.Regular`.

2.2.2.2 Authentic token hijacking

The following attacks aim at extracting value from treasury pools using authentication tokens. Spending treasury pools require that a transaction input carries an authentication token (of the right asset class). In practice this means that unstaking a position is the only way to get value from the pools (in order to pay for the staking rewards). To avoid possibly exceeding the size limit on validator scripts, the asset class in question is not a script parameter but part of the datum in treasury pools.

We considered hijacking authentic tokens an interesting vector of attack to explore for the following reasons:

- The treasury pool validator only checks for an authentic token. Hence the possibility to empty pools if we manage to steal or duplicate a genuine token.
- The asset class of authentic token is part of the datum, not a script parameter, so it might be possible to modify it.
- The life cycle of authentic tokens from minting to abandonment in the treasury pools themselves is complex and uncommon enough to deserve closer inspection.

The contract **is robust against** the following attempted attacks, which are presented in module `Audit.Staking.Locked.AuthTokenHijacking`:

Rob a treasury pool with an authentic token from another
In trace `traceHijacking`, a staker creates their own treasury pool in order to host the authentic token of a staking position after unstake. Then, they try to use that additional treasury pool (which holds an authentic token) to rob the MELD tokens from an official MELD treasury. This fails because the treasury pool validators prevents this by checking whether the authentic token comes from a UTxO with same datum (the datum is the same among all treasury pools associated to a given authentic token asset class).

Swap authentic tokens of different classes during double unstake
In trace `(traceTwoDifferentAuthTokens True)`, there are two kinds of treasury pools associated to two different authentication tokens. Two staking positions are opened, each associated to one of the asset classes. This attack consists in unstaking both position in the same transaction and swap the authentic tokens in the meantime. After this, the tokens in the treasury pools would match the datum of the other pool and the check that made the previous attack fail would pass because the pools have different datum (two different asset classes).

Fortunately, such a double unstaking fails when one tries to swap authentic tokens. Indeed, function `validateLockedStaking` checks that the authentication tokens are all in treasury pool with the expected datum after unstaking.

Change the asset class of the treasury script when unstaking
In trace `traceAttemptChangeTreasuryAssetClass`, a staker unstakes after the unlock time. They transfer the authentic token of the staking position to a treasury pool and, in the same transaction, try to change the asset class in the datum of the treasury pool to an adversary asset class in order to retrieve value from the treasury pool later.

Function `validateLockedStaking` prevents the modification of the datum. Indeed, the test called "Must withdraw the correct interest" checks that the meld tokens that were in the treasury pool remain in a treasury pool with the same datum (apart from the interest which are withdrawn).

Steal an authentic token during a double unstake
In trace `traceAttemptAuthTokenTheftInDoubleUnstake`, a staker opens two identical staking positions. Two authentic tokens are minted and locked in each of the positions. After the unlock time, the staker unstakes both positions in the same transaction but only sends one of the authentic tokens to a treasury pool and tries to steal the other.

Condition "Auth tokens must be at the correct locations" of the validator script fulfills its role and prevents such a grabbing of authentication token.

Unstake two positions with only one authentic token
In trace `traceAttemptDoubleUnstakeWithOneAuthToken`, a staker opens a legit position that mints an authentic token and also an illegitimate one manually. They then try to unstake both in the same transaction and obtain reward for both the legitimate and illegitimate positions.

Here condition "Must withdraw the correct amount" checks that the amount which went out of the treasury pools is exactly the interest expected by one of the position, meaning that it is impossible to unstake two positions simultaneously, no matter the presence of authentication tokens.

2.2.2.3 Experiments with NFT evolution

As stated in section 2.1, Tweag assumed the existence of a minting policy that takes care of burning a lower level NFT to mint higher level one. This section details experiments on NFT evolution disregarding the possible attacks involved by having more than one level of an NFT on the chain.

Undeclared NFT in staking position *Severity: Low*

It is possible to open a staking position whose value contains an NFT that is not officially declared in the datum `dNFTTokenName = Nothing`. See `traceUndeclaredNFT` in module `Audit.Staking.Locked.Evolution`.

The example traces in the module show that:

- it is impossible to upgrade the NFT to a higher level
- it is impossible to obtain the NFT bonus when unstaking
- it is possible to unstake (without the NFT bonus)

Out of bounds levels *Severity: Low*

Official levels for the NFT are 1, 2, 3, 4, and 5. It is assumed that the existing minting policy forbids mintng other levels. Note that validator `validateLockedStakingEvolveNFT` double checks that the level of the NFT is within the bounds. On the contrary, no such check is performed at the creation of the staking position, since `validateLockedStakingAuthToken` accepts any token name. Especially, it is possible to give an NFT of level 0 to a newly created staking position. This NFT is functional, in the sense that it is possible to evolve it later from the illegal level 0 to the legitimate level 1.

2.2.2.4 Additional remarks

Possibility to unstake two positions associated to different authentication asset classes in the same transaction *Severity: Low*

In module `Audit.Staking.Locked.AuthTokenHijacking` (see `(traceTwoDifferentAuthTokens False)`), two staking positions using authentication tokens belonging to different asset classes are closed (unstake) simultaneously. It is noticeable that this situation makes it possible to unstake two valid positions in the same transaction. This is probably an unintended behaviour since several tests in the validator script actively check against multiple unstaking.

Conditions on the validator check more than what is declared *Severity: Medium*

Validator `validateLockedStaking` for `Unstake` case contain 5 conditions. Three of them check what their name suggest: signature ("Must be signed by owner"), time and quality of the inputs ("Position still locked" and "Cannot consume more than required treasury").

The last two check more than suggested by their names. "Must withdraw the correct amount" checks not only the amount withdrawn, but also:

- checks that the datum of the treasury pool is not modified,
- guarantees than only one withdrawal is performed by the transaction, since the amount of Meld tokens leaving the treasury pools must correspond exactly to the interest amount of one staking position. (With the irrelevant exception that it is possible to unstake simultaneously two positions which both stake 0 melds.)

Similarly "Auth tokens must be at the correct locations" also checks that there is only one input which is not a treasury pool and contains an authentication token. Thus, it prevents multiple withdrawals in the same transaction.

The impossibility to unstake several positions in one transaction is crucial to prevent some of the attacks we tried. Yet, it is undocumented and not clearly listed as an item checked by the validator. There is a non-negligible risk of accidentally removing that property in future updates of the validator.

■ **Nested conditions in the validator scripts would benefit from more tracing** *Severity: Lowest*

In function `validateLockedStakingAuthToken`, the verification of the validity of the output is performed via a function `checkOutput` which performs 3 different checks. Providing information about the failing one would be helpful, when a transaction fails.

It can be noted that this issue is also present in the validator `validateVariableStakingAuthToken`, where the sub-function `checkOutput` does 5 distinct checks.

■ **The requirement on the useless treasury inputs makes it harder to collect remainders** *Severity: Low*

The validator `validateLockedStakingAuthToken` performs the "Cannot consume more than required treasury" check. This check ensures that it is impossible to use only a strict subset of the treasury pools given as input to the transaction. This means that whenever a treasury pool contains a very small amount of meld, it is not easy to simply empty it and use a treasury pool with a large amount to complete the transaction. One has to wait for the existence of sufficiently many almost empty treasury pools to consume and merge all of them in the same transaction. This impossibility to do some cleaning on the fly might become annoying.

Furthermore, we did not identify any issue related to the fact of allowing transactions consuming more treasury pools than required.

2.2.3 Variable Staking contract

This contract is the most advanced of the three. Since the client only provided it two days before the end of the audit, we did not have enough auditing time to draw conclusions about the correctness of the design and the implementation.[1]

2.2.3.1 Basic Usage

The most basic use of the contract consists in staking some Melds and withdrawing it (all or part of) at some point in the future along with the interest amount.

The traces of module `Audit.Staking.Variable.Regular` test those basic uses of the contract. So, in all those traces, a staking position is opened, later a withdrawal request is sent, and finally this request is either fulfilled or canceled. In each of the cases presented, we verified that an evolution of the interest rate of the treasury pool can happen at any point of the trace without causing issues. All those basic traces validate without trouble.

[1] Hence, this section only presents basic traces and one issue about documentation. This small amount of issue is purely a consequence of the small amount of time spent to explore the contract.

2.2.3.2 Remarks on Documention

The Computation of the interest amount is complex and undocumented *Severity: Low*

When a financial position has an interest rate, one often expects the total interest amount to be the principal multiplied by the rate and the time the position has remained opened.

The client has made a completely different choice which does not involve any computation with time. This strategy to compute the interest is not properly documented and was only understood after an exchange of messages between the auditing team and the client.

The treasury pools come with an interest rate, which is copied within the field `dMarketInterestRate` of the staking position. On a regular basis, the interest rate stored within the treasury pools is raised by the administrators, and the interest a user benefits from when they unstake a position, is the difference between the interest at the time they opened the position (the `dMarketInterestRate`) and the one stored in treasury pools.

So, the field `dInterestRate` of the treasury pools contains the interest someone would get if they opened their position the first day. And the field `dMarkedInterestRate` of the datum of the staking position store how much of those interest rate were already consumed (*i.e.* either missed because the position is not opened since the very beginning, or already considered in the `dInterestAmount` field of the staking position).

2.3 Conclusion

We focused the audit on the *Vesting* contract and the *Locked Staking* contract. We have not found any vulnerability or important (apart from irrelevant edge cases) violations of the specifications. We attempted a family of attacks against *Locked Staking* to steal assets from treasury pools. They involve closing several locked staking positions in the same transactions. Multiple unstaking in the same transaction is not allowed by the validators but there is no clear dedicated guard in the source code. We suggest that the guard be made explicit in order to avoid weakening it accidentally in future update. The source code is very clear and well-commented. So is the official documentation. Finally, we note that many security aspects of the contracts, in particular *Vesting* and the NFT management for *Locked Staking*, heavily rely on off-chain procedure carried out by trusted administrators. These procedures and administrators are probably, to our knowledge, the weakest links in the security of the audited smart contracts.