TWEAG
by Modus Create

Technical Review of Wolfram Oracle

Tweag - Modus Create
January 24, 2025

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 Wolfram Labs'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 Wolfram Labs. 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 script of the Wolfram Oracle protocol as of commit `737f7e..ecaf3e` 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. Among those files, `Types.hs` consists of copy-pasted code from the Marlowe project, and `Utils.hs` consists of function used to export the validators, thus our analysis was mostly focused on `OracleValidator.hs`, which contains the actual validator code. These files have been very slightly patched to account for mismatched dependencies, as depicted on appendix A.1.

`sha256sum`	File Name
`44f2d0..63308f`	`OracleValidator/src/Types.hs`
`a5a9d2..42e8b7`	`OracleValidator/src/OracleValidator.hs`
`a6ec11..9c0ef8`	`OracleValidator/src/Utils.hs`

Table 2.1: *Files analyzed and their sha256 sum*

Our analysis was based on the original documentation provided by Wolfram Labs and on oral and Slack conversations with Wolfram Labs. The relevant documentation files are listed in Table 2.2 and their contents will be referred to as *the specification* of the protocol.

`sha256sum`	File Name
`6af82e..9568db`	`MilestoneOne/README.md`
`5d34e1..ca3ac5`	`MilestoneTwo/WolframOracleHarvester.ipynb`

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

Tweag used `cooked-validators` (`https://github.com/tweag/cooked-validators`) to conduct testing over Wolfram Oracle, a home-made open-source tool dedicated to interacting with smart contracts.

2.2 Delivery

This report is accompanied with a Haskell project comprising:

- A nix environment containing the required dependencies of the project.
- A custom version of our `cooked-validators` library[2] adapted for Wolfram Labs's validators.

[1] `https://github.com/WolframBlockchainLabs/MarloweSC-Execution`

[2] 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.

- A test suite of Wolfram Oracle. This test suite emphasises both regular use cases of the product, as well as tweaked use cases that may surface vulnerabilities.
- An example of a fixed validator, written from the set of remediation suggestions comprised in this report. It is important to note that this fixed validator is provided as a courtesy to Wolfram Labs and is to be used as *an example* for later remediation. Tweag does not provide any guarantee regarding the full correctness of this example validator. This fixed validator is also given in appendix A.2, for the purpose of being referenced in remediation suggestions.
- A test suite of this fixed validator, emphasising the correction of uncovered issues.

This project is meant to be used by Wolfram Labs for remediation purposes. The test suite in particular is meant to be used with later iterations of the product to showcase the correction of findings. A `README.md` file is provided within the project to ensure Wolfram Labs can properly use and run the project.

2.3 Wolfram Oracle

Wolfram Oracle is a smart contract comprising a single validator to be used with the `Spending` purpose. In other words, it is a validator which is meant to own UTXOs and control how they can be spent. We call this validator the *Oracle validator*.

This smart contract is meant to provide extraneous information about real-world currencies rates to a Marlowe contract. Marlowe is an existing DSL to write smart contracts and an on-chain interpretor of those smart contracts. Marlowe contracts are stored in datums and their state can evolve through various transactions. In particular, choices can be made over a Marlowe contract, using a specific redeemer to consume the associated UTXOs.

To use *Oracle validator*, one must first pay a certain datum at the validator's address. Then, two use cases are supported, for which we have given custom names for later references:

- *Execute* is the main use case. It allows the consumption of the oracle UTXO alongside a Marlowe contract, for which a choice is currently being executed. The oracle is meant to provide information to guide that choice. In that case, the validator checks that the Marlowe input is present and consumed with a redeemer that corresponds to the expectation of the oracle's datum. It is also checked that this transaction is executed before a certain deadline.
- *Reclaim* is the secondary use case. It is meant to avoid unspendable UTXOs by allowing the oracle UTXO to be redeemed by a certain peer after the deadline has passed and the first use case has not been successfully triggered. In this case, it is checked that the deadline has passed and the authorized peer has signed the transaction.

The high-level set of constraints handled by the *Oracle validator* is thus as followed:

- The deadline has passed.
- A specific peer has signed the transaction.
- The Marlowe input is present in the transaction.
- The Marlowe input is consumed for a single choice purpose.
- This choice corresponds to the expected choice of the oracle.

2.4 Findings

Table 2.3 lists our concerns with the current Wolfram Labs's code base based on our partial exploration during a limited period of time.

Severity	Section	Summary
Critical	2.4.1.1	Incorrect validity interval check
Critical	2.4.1.2	Incorrect top-level exclusive disjunction condition
High	2.4.4.1	Lack of Wolfram wallet in the codebase
Medium	2.4.2.1	Incomplete pattern matching on the Marlowe redeemer
Medium	2.4.2.2	Unchecked warnings and lack of coding tooling
Medium	2.4.2.3	Unnecessary copy-paste of the Marlowe codebase
Medium	2.4.3.1	Lack of deserialization optimization
Low	2.4.2.4	Improvable variable names
Low	2.4.2.5	Too many variables and types in the `where` clause
Low	2.4.2.6	Incomplete error messages
Low	2.4.2.7	Outdated dependencies
Low	2.4.3.2	Oracle datum is not required to be inlined
Lowest	2.4.2.8	Unnecessary unfolding of `TxOutRef` in `UTXODatum`
Lowest	2.4.2.9	Leftover of dead code

TABLE 2.3: *Table of findings*

Throughout the rest of this section, we will detail each of our findings.

2.4.1 Logical flaws

2.4.1.1 Incorrect validity interval check

Severity: Critical

Issue In the validator, the only condition comparing the deadline to the validity interval is the following expression: `contains (from (limitTime + 1)) txValidRange`. This checks that the validity interval is included in $]$`deadline`$, +\infty[$ meaning that we are currently past the deadline.

In the *Reclaim* case, this condition is checked directly and matches the requirement that the deadline must have passed.

In the *Execute* case, the validator relies on the fact that for the top-level XOR in `xorConditions` to be `true`, the conditions for the *Reclaim* case `bCondition` must be false. That is, the validator relies on the **negation** of the above condition to hold as an indirect check that the deadline hasn't passed yet.

However, checking for the negation isn't sufficient, because $\neg(I \subset]$`deadline`$, +\infty[)$ is in fact **weaker** than the desired condition $I \subset]-\infty, \text{deadline}]$. Indeed, the negated condition is equivalent to $I \cap]-\infty, \text{deadline}] \neq \emptyset$. Any validity interval that starts at `deadline` or before but might potentially end much later will satisfy the negated condition and pass the *Execute* case validation. Typically, choosing $]-\infty, +\infty[$ as a validity interval allows to *Execute* at an arbitrary moment even after the deadline.

In our test suite This behaviour is showcased in trace *executeAfterDeadline* in which we set the validity interval to $]-\infty, +\infty[$ and expect a failure while getting a success, in line 27 of file `Audit.hs`.

Remediation We suggest to implement a direct condition to enforce that the *Execute* action is performed before the deadline. An example of such condition can be written similarly to the existing condition for *Reclaim*: `contains (to limitTime) txValidRange`. Our example validator defines this explicit condition on line 196 and uses it on line 188.

2.4.1.2 ■ Incorrect top-level exclusive disjunction condition

Severity: Critical

Issue The implementation of *Oracle validator* decides which action is being performed by the user by computing two boolean conditions, called `aCondition` and `bCondition`. Those conditions are supposed to correspond respectively to the requirements of the *Execute* case and the *Reclaim* case. They are assumed to be exclusive: either exactly one of the two is true, and thus the corresponding case is allowed by the validator, or the validator rejects the transaction. This is why the validator top-level check is `aCondition `xOr` bCondition`.

`bCondition` itself is a logical conjunction of two separate conditions. Let `aCondition` be A and `bCondition` be $B \wedge C$, where:

- (A) The redeemer of the Marlowe contract input has the right shape and its choice id matches the choice name in the oracle's datum
- (B) The validity interval of the transation is contained in $]$deadline$, +\infty[$
- (C) The beneficiary after the deadline has signed the transaction

Below is a truth table of the validator, depending on the possible individual values of A, B and C.

case	A	B	C	$B \wedge C$	$A \oplus (B \wedge C)$
a	0	0	0	0	0
b	0	0	1	0	0
c	0	1	0	0	0
d	0	1	1	1	1
e	1	0	0	0	1
f	1	0	1	0	1
g	1	1	0	0	1
h	1	1	1	1	0

According to this truth table, we can see that there currently are 4 cases of validation failure (a, b, c and h), and 4 cases of success (d, e, f and g). The failure cases behave according to the specification. The success cases however need to be looked at more closely:

- (d) Normal case of *Reclaim* after the deadline by the rightful beneficiary
- (e) Supposedly normal case of *Execute*. However, because $\neg B$ doesn't entail that we are necessarily before the deadline, this is not always a valid case; see 2.4.1.1.

- (f) Variant of (e) where the beneficiary has signed the transaction.
- (g) An arbitrary user that is not the beneficiary after the deadline can *Execute* after the deadline simply by signing the transaction. Indeed, even if `B` is true, this alone isn't sufficient to make `B && C` true and make the whole transaction fail because `C` is false.

As a summary, case (e) and (f) allows to unduly *Execute* after the deadline by setting a validity interval that starts before the deadline, as explained in 2.4.1.1. Case (g) poses a different issue, relying on the fact that merely signing the transaction by someone else that the beneficiary after deadline ensures that `bCondition` is false independently from the value of the deadline check `B`, which in effect entirely bypasses the deadline check.

In our test suite This behaviour is showcased in trace *executeAfterDeadline'* in which we set the validity interval to start 100 slots after the deadline, while signing with someone else than the beneficiary, and expect a failure while getting a success, in line 28 of file `Audit.hs`.

Remediation The main source of the issue here is that both validation cases are attempted to be treated with the single redeemer `()`. This is an anti-pattern where, instead of having a significant redeemer to inform the validator which case should be checked, the transaction itself should somehow bear that information. Instead, there should be as many redeemers as possible cases to redeem the UTXO. We suggest to implement a dedicated redeemer type as follow:

```
data Oracleedeemer = Execute | Reclaim
```

Then, the validator should pattern-match over this redeemer and conduct only positive checks required for each of those cases instead of relying on the negation of other tests. Our example validator defines an explicit redeemer on line 49 and case splits over it on lines 170 and 185.

2.4.2 Code quality

2.4.2.1 Incomplete pattern matching on the Marlowe redeemer

Severity: Medium

Issue Although GHC notifies it as a warning instead of an error by default, Haskell requires that pattern matching is exhaustive and that functions are total. A non-exhaustive pattern matching is often the indication of a mistreatment of a data structure, with forgotten cases for which the behaviour is unspecified.

The *Oracle validator* contains such a non-total occurrence, when matching over the redeemer of the Marlowe contract input. It expects a specific shape, namely a list with exactly one element that must also satisfies additional constraints, but doesn't handle cases when the redeemer is a list with e.g. two elements, or no elements at all.

While this doesn't affect the boolean result of the validator, since the partial compiled code will raise an error, non exhaustive pattern matching is dangerous and detrimental. Indeed, it is unclear whether the lack of all cases is intentional or not, thus leaving room for unexpected coding error. Additionally, if the redeemer hits one of the non-covered cases, the contract fail without any proper error message, leaving the transaction author unaware of what went wrong. Overall total functions are always to be preferred to avoid unexpected runtime errors.

In our test suite This behaviour is showcased in trace *executeMultipleInputsInRedeemer* for which we expect to have the error message `Redeemer Match: False` but instead have no error message at all, which explain why our test fails on line 20 of file `Audit.hs`.

Remediation We advise to either manually handle the case when the redeemer has 0 or more than 1 elements, or to use a catch-all clause `_` at the end of the pattern matching. Our example validator handles this case through a catch-all clause on line 210.

2.4.2.2 Unchecked warnings and lack of coding tooling

Severity: Medium

Issue We encountered various warnings and formatting inconsistencies when reviewing the provided code, hinting at the fact that perhaps no standard Haskell tooling has been used during development.

Remediation We advise to use a code formatter such as Ormolu, an LSP client such as haskell-language-server and to set the compilation flag `-Werror` by default to turn warnings into errors when working with production code. Our example validator has been formatted using Ormolu 0.7.2.0.

2.4.2.3 Unnecessary copy-paste of the Marlowe codebase

Severity: Medium

Issue An entire file, `Types.hs`, is a copy-pasted version of data types coming from Marlowe. The only purpose of this file is to provide the **`Input`** type to be used as a redeemer for the Marlowe input. The file is lengthy, because this type has a significant depth and involves many other definitions. However, the codebase only require one field from this redeemer, the name of the choice id. This entails a doubling of the size of the project, while increasing the size of the *Oracle validator* for very little actual dependency.

Remediation Data is stored on-chain as **`BuiltinData`** which can be deserialised at will in a custom manner without requiring the full type of the expected data. Thus, we suggest to avoid this copy-pasting by removing the `Types.hs` file altogether from the codebase. Instead, we suggest to implement a minimal data structure only encompassing the used fields of the redeemer, and deserialising said redeemer to that data structure. This mechanism is described thoroughly in issue 2.4.3.1 and our example validator makes use of and defines the necessary constructs for the custom Marlowe redeemer from line 62 to line 102.

2.4.2.4 Improvable variable names

Severity: Low

Issue Some of the top-level, important conditions of the validator are given names that aren't semantically meaningful, such as `xorConditions`, `aCondition` and `bCondition`. Using semantically meaningful names helps with reviewing the code, and to make sure that the semantics of the variable and its implementation do agree.

Remediation We suggest to ensure the names of the project variables are meaningful. Each condition in our validator is named relevantly. For example, on line 196 we define `beforeDeadline` which does ensure the deadline has not passed when running the *Execute* transaction.

2.4.2.5 ■ Too many variables and types in the `where` clause

Severity: Low

Issue The *Oracle validator* introduces many variables derived from either the datum or the script context. While the *Oracle validator* is rather small, it introduces a total of 14 variables in the **`where`** clause on line 62. Admittedly, introducing intermediate variables often helps understanding and modularize, but an excess of small variables can make it harder to follow the information flow.

Additionally, while it's considered good practice to use type annotations for top-level declarations in a Haskell module, these type annotations should often be avoided for local variables, unless they are particularly meaningful. Indeed, such type annotations quickly clutter the code and can easily be queried through the use of an LSP client if needed. The oracle validator uses such superfluous type annotations on the variables introduced by the **`where`** clause.

Remediation Most of those variables can be obtained by destructing the datum and the script context directly in the arguments, which we advise because it is more concise, readable and idiomatic. This also has the upside of avoiding the unnecessary typing information altogether. For example, this simple destruction of the **`UTXODatum`**:

```
(UTXODatum transactionID inputIndex choiceName _ limitTime beneficiary)
```

replaces this whole block of code:

```
transactionID :: TxId
transactionID = transactionId datum

inputIndex :: Integer
inputIndex = transactionIndex datum

choiceName :: Builtins.BuiltinByteString
choiceName = choiceGivenName datum

limitTime :: POSIXTime
limitTime = deadlineLimit datum

beneficiary :: PubKeyHash
beneficiary = beneficiaryAfterDeadline datum
```

Our example validator only used meaningful local variables and does not explicitly state their types. Instead, it uses relevant names or type application to ensure the types are known. Additionally, using a language server client allows us to know the type of any variable during programming.

2.4.2.6 ■ Incomplete error messages

Severity: Low

Issue The *Oracle validator* run can fail for more reasons than meet the eyes. In addition to the constraints about the deadline and signatories, the retrieval of the choice name within the Marlowe input is in fact the conjunction of many requirements:

a) The Marlowe input must be present in the transaction.

b) The Marlowe input must be consumed with a redeemer, i.e. not be a private key output.

c) The redeemer must resolve as a list of inputs.

d) The list of inputs must be a singleton.

e) The singleton input must have an input content resolving to a choice.

f) This choice must have the same name as the one stored in the oracle datum.

The error messages in the code do not reflect this complexity. Instead, a single message is traced *Redeemer Match: False* (or no message at all, in the case of a redeemer with several inputs, see 2.4.2.1) which bears very little information as to what actually failed among those checks. Similarly, the checks about the signatory and the deadline are grouped and subject to a single error message. This makes very hard to know what went wrong during testing, and is overall detrimental to the clarity of the codebase.

Remediation We suggest to treat each requirement separately and have dedicated error message to most, if not all, of the validator failure cases. Note that, in production, the default behaviour of the Plutus compiler is to erase any tracing, thus they will not increase the validator's size. In our example validator, most of the failure cases have their own separate error message. In the case of the redeemer constraints, this is handled using an **`Either`** monad which stores the proper error message, from line 200 to line 210.

2.4.2.7 ■ Outdated dependencies

Severity: Low

Issue The Wolfram Oracle dependencies on various Plutus and Cardano APIs are old and can be upgraded without much effort nor changes to the code. While newer versions of Plutus aren't strictly required to make this script work, keeping dependencies up-to-date is good practice and could also unlock new capabilities, should the script evolve, such as multi-purpose scripts (see 2.4.3.2).

Remediation We suggest to update the dependencies to the most recent that do not induces changes into the codebase, and later on consider some more updates depending on the evolution of the project. Our example validator uses `plutus-ledger-api` version 3, before the Chang hard fork which introduces multi-purpose scripts. This has also been patched in the codebase to have the *Oracle validator* interface properly with `cooked-validators` (see A.1).

2.4.2.8 ■ Unnecessary unfolding of `TxOutRef` in `UTXODatum`

Severity: Lowest

Issue The oracle datum stores both `transactionId` and `transactionIndex` separately but always use them together to identify the Marlowe contract **`TxOutRef`**.

Remediation We suggest to directly store a **`TxOutRef`**, as we did in our example validator on line 28.

2.4.2.9 ■ Leftover of dead code

Severity: Lowest

Issue Some code is commented out on line 93, which seems to correspond to an old way of ensuring the Marlowe UTXO appears in the inputs of the transactions. It is not clear if it is dead code that should be removed or code that needs to be enabled later.

Remediation We recommend to get rid of this dead code. Additionally, we suggest to improve the quantity of comments present in the validator code. Everything that is non-trivial should be commented, as we showcase in our example validator.

2.4.3 Design flaws

2.4.3.1 Lack of deserialization optimization

Severity: Medium

Issue When processing a transaction, the Cardano Ledger is expecting that this transaction will provide a certain amount of fees. These fees are correlated, among other things, to the size and resource consumption of the involved script, both in terms of processing cycle and memory requirements. Thus, conduction optimisation to have a script that is smaller, runs faster or uses less memory has a direct impact of making each transaction involving the script cheaper. In particular, it is known in the Cardano community that the default way of deserialising the script context and sometimes the redeemer and datum can make up for a good portion of the total runtime cost of the script and thus can drastically be optimized.

The oracle validator deserialises four data elements initially provided as **`BuiltinData`**:

- The redeemer of the *Oracle validator*
- The **`UTXODatum`**
- The script context
- The redeemer of the Marlowe input

While the redeemer of the *Oracle validator* is just `()` and all but one fields of the **`UTXODatum`** are used, there is performance to be gained by optimizing the deserialisation of the script context and Marlowe redeemer. The idea between the usual optimisation is to only deserialise fields that are actually used by the validator, and leave other fields as **`BuiltinData`**, thus preventing them from being deserialised. This is achieved by creating views over the needed data types, essentially copying their Haskell definition and changing the type of any unused record field to **`BuiltinsData`**, thus saving the cost of deserialising them. The technique is demonstrated in the Tweag Oracle Validator example provided with this report. In the case of the Marlowe redeemer, in addition of the usual gain this technique provides, there is another upside in Wolfram Oracle: by only deserialising the redeemer to the expected form, we let the deserialisation conduct preliminary checks for us, and ensure the validator receives a redeemer of the right shape. Additionally, this fixes issue 2.4.2.3 by removing the need for the `Types.hs` copy-pasted file.

Remediation We suggest to implement views over the script context and the Marlowe redeemer to ensure only the required fields are deserialized, by leaving unused fields as **`BuiltinData`**. Our example validator does include such optimization. The Marlowe redeemer has been optimized as shown on issue 2.4.2.3 while the script context optimization occurs between line 104 and line 148.

2.4.3.2 Oracle datum is not required to be inlined

Severity: Low

Issue Oracles (in the general sense) are meant to store pieces of information from the real world on-chain, typically trade rates. This information is often expected to be visible and accessible by anybody to ensure these rates are correct and properly used in smart contracts relying on them. However, in the current version of Wolfram Oracle, the **`UTXODatum`** can be stored hashed, thus forbidding users to have access to its content and take actions accordingly without any external off-chain, un-hashed, version of the datum. Since Plutus version 2, datums can be stored inline within UTXOs, thus ensuring users can have access to their content, which is not enforced in Wolfram Oracle. Additionally, the current

version of the *Oracle validator* does not impose any specific peer to run the *Execute* transaction, which can be triggered by anybody. If there were any such specific peer (see 2.4.4.1), assuming they have access off-chain to the whole datum to provide to the transaction would be more understandable.

Remediation There are two options here. The first option is to acknowledge that the datum can in fact be hashed and document that the users expected to run the *Execute* transaction will indeed have access to the right datum off-chain, for the purpose of providing the datum to the transaction body. The second option is to enforce that the datum is paid inlined and not hashed in the initial transaction. This second version however cannot be enforced without significant changes to the validator. More precisely, as the validator is not invoked during the initial payment, such verification could only be made through a minting policy associated with the current script, that would mint an NFT while ensuring the datum is inlined and has the right shape (this could also be used to ensure, for instance, that the deadline has not already expired when the initial payment is issued). This minting policy would typically be a second purpose added to the script, and thus would induce a certain amount of additional on-chain code. This change is not showcased in our example validator.

2.4.4 Unclear specification

2.4.4.1 ■ Lack of Wolfram wallet in the codebase

Severity: High

Issue The documentation provided by Wolfram Labs mentions an additional party in a diagram representing the nominal *Execute* transaction, named Wolfram wallet, where the funds of the oracle should be redirected to. There is no mention of this wallet in the code, and indeed the *Execute* case doesn't impose any specific output nor signer, meaning that an attacker can execute the transaction and redirect the remaining funds to their own wallet (and do so repeatedly by regularly querying the chain for new Oracle UTXOs, thus generating a profit).

Remediation The role of Wolfram wallet, if any, needs to be properly documented and the implementation needs to be adapted accordingly. For example, the beneficiary wallet could be used as Wolfram wallet and be required as a signer for the *Execute* transaction as well. If this is not satisfactory, the Wolfram wallet could either be put in the datum, by a parameter of the script, or be directly hard-coded into the validator, to enforce it receives the right funds. In any case, this situation needs to be clarified. This is not showcased in our example validator, as it depends on the expected semantics of the contract.

2.5 Conclusion

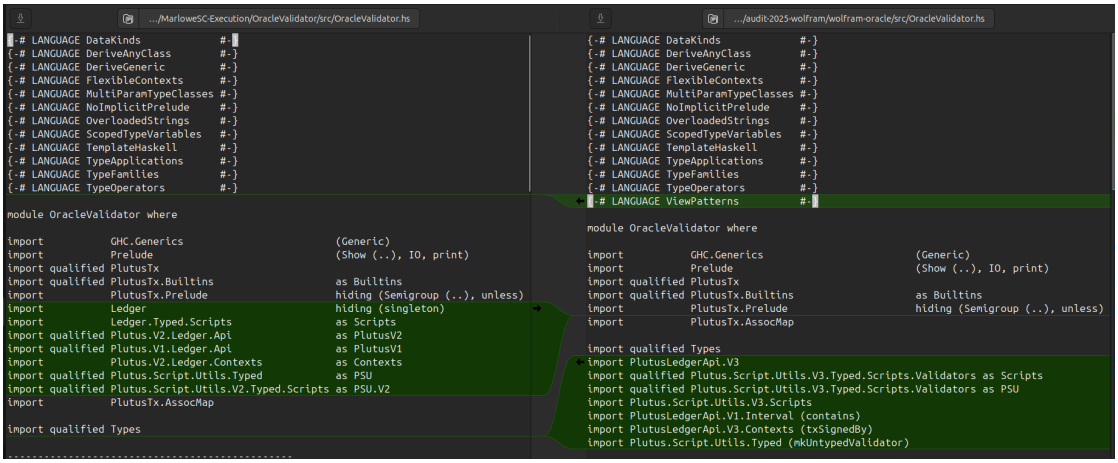
This report outlines the 14 concerns that we have gathered while inspecting the design and code of Wolfram Labs, pertaining to the code contained in the files listed in Table 2.1. As stated in Chapter 1, Tweag does not recommend for nor against the use of any work referenced in this report. Nevertheless, the existence of *high* and *critical* severity concerns is a warning sign.

Appendix A

Appendix

A.1 Patch to *Oracle validator*

In order to have *Oracle validator* interface properly with `cooked-validators`, a small patch has been implemented. This patch solely revolves around upgrading some dependencies in *Oracle validator* and does not change the validator's behaviour in any way. Here is the diff between the two files headers:



In addition to this top-level change, the qualification of the imported elements have been revise to match the top-level qualifications of the import. Otherwise, the file is unchanged. The other two files have very little changes as will, with `Utils.hs` subject to the most changes. However, this file does not provide any on-chain component and thus is not relevant in the on-chain code behaviour.

A.2 Tweag Oracle Validator

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ViewPatterns #-}
{-# LANGUAGE NoImplicitPrelude #-}

module Audit.TweagOracle where

import Cooked.Validators qualified as C
import Plutus.Script.Utils.Scripts qualified as Script
import Plutus.Script.Utils.V3.Typed.Scripts qualified as Scripts
import Plutus.Script.Utils.V3.Typed.Scripts.Validators qualified as Script
import PlutusLedgerApi.V1.Interval qualified as Api
```

```haskell
import PlutusLedgerApi.V3 qualified as Api
import PlutusTx qualified
import PlutusTx.AssocMap qualified as PlutusTx
import PlutusTx.Prelude
import Prelude qualified as P

-- * Tweag oracle datum and associated instances

-- The main differences between TweagOracleDatum and UTXODatum are
-- - names have been changed
-- - a TxOutRef combines the txId and index
data TweagOracleDatum = TweagOracleDatum
  { marloweUtxo :: Api.TxOutRef,
    choiceName :: BuiltinByteString,
    dataTag :: BuiltinByteString,
    deadline :: Api.POSIXTime,
    beneficiary :: Api.PubKeyHash
  }
  deriving (P.Show, P.Eq)

-- PlutusTx.Eq instance, needed by cooked-validators
instance Eq TweagOracleDatum where
  (==) = (==)

-- We leave it to Plutus to derive the BuiltinData representation of
-- TweagOracleDatum
PlutusTx.unstableMakeIsData ''TweagOracleDatum

-- * Tweag Oracle Redeemer and associated instances

-- TweagOracleRedeemer provides two redeeming cases for the oracle contract
-- - Execute a Marlowe choice with the info provided by this oracle
-- - Reclaim the oracle funds after the deadline has passed
data TweagOracleRedeemer
  = Execute
  | Reclaim
  deriving (P.Show, P.Eq)

-- PlutusTx.Eq instance, needed by cooked-validators
instance Eq TweagOracleRedeemer where
  (==) = (==)

-- We leave it to Plutus to derive the BuiltinData representation of
-- TweagOracleRedeemer
PlutusTx.unstableMakeIsData ''TweagOracleRedeemer

-- * Custom Marlowe Redeemer

-- We only redefine parts of the Marlowe redeemer that are relevant to our
-- validator. This is limited to ChoiceId, InputContent and Input. The rest can
-- be left out as BuiltinData. This will allow to perform a first round of
-- verification over the shape of the redeemer.

data ChoiceId = ChoiceId BuiltinByteString BuiltinData
  deriving (P.Eq, P.Show)

-- We use the safe way of building the BuiltinData representation, using
-- dedicated indexes for constructors.
PlutusTx.makeIsDataIndexed ''ChoiceId [('ChoiceId, 0)]
```

```
75
76  -- PlutusTx.Eq instance, needed by cooked-validators
77  instance Eq ChoiceId where
78    (==) = (==)
79
80  data InputContent
81    = IChoice ChoiceId BuiltinData
82    deriving (P.Eq, P.Show)
83
84  -- Important: We need to keep the same index as the Marlowe representation,
85  -- which is here 1 instead of 0, because there are other cases in the initial
86  -- representation which we did not need and thus discarded.
87  PlutusTx.makeIsDataIndexed ''InputContent [('IChoice, 1)]
88
89  -- PlutusTx.Eq instance, needed by cooked-validators
90  instance Eq InputContent where
91    (==) = (==)
92
93  data Input
94    = NormalInput InputContent
95    | MerkleizedInput InputContent BuiltinData BuiltinData
96    deriving (P.Eq, P.Show)
97
98  PlutusTx.makeIsDataIndexed ''Input [('NormalInput, 0), ('MerkleizedInput, 1)]
99
100 -- PlutusTx.Eq instance, needed by cooked-validators
101 instance Eq Input where
102   (==) = (==)
103
104 -- * Custom Tweag script context
105
106 -- TweagTxInfo is basically a copy of the TxInfo record from plutus-ledger-api
107 -- where the types of the fields that won't be needed by this validator have
108 -- been replaced by `BuiltinData`. Doing so, these fields won't be deserialized
109 -- uselessly. In some sense, TweagTxInfo is a partial view of TxInfo.
110 data TweagTxInfo = TweagTxInfo
111   { txInfoInputs :: BuiltinData,
112     txInfoReferenceInputs :: BuiltinData,
113     txInfoOutputs :: BuiltinData,
114     txInfoFee :: BuiltinData,
115     txInfoMint :: BuiltinData,
116     txInfoTxCerts :: BuiltinData,
117     txInfoWdrl :: BuiltinData,
118     txInfoValidRange :: Api.POSIXTimeRange,
119     txInfoSignatories :: [Api.PubKeyHash],
120     txInfoRedeemers :: PlutusTx.Map Api.ScriptPurpose Api.Redeemer,
121     txInfoData :: BuiltinData,
122     txInfoId :: BuiltinData,
123     txInfoVotes :: BuiltinData,
124     txInfoProposalProcedures :: BuiltinData,
125     txInfoCurrentTreasuryAmount :: BuiltinData,
126     txInfoTreasuryDonation :: BuiltinData
127   }
128   deriving (P.Eq)
129
130 -- PlutusTx.Eq instance, needed by cooked-validators
131 instance Eq TweagTxInfo where
132   (==) = (==)
133
```

```
PlutusTx.unstableMakeIsData ''TweagTxInfo

-- TweagScriptContext is a partial view of the ScriptContext type from
-- plutus-ledger-api. See TweagTxInfo.
data TweagScriptContext = TweagScriptContext
  { scriptContextTxInfo :: TweagTxInfo,
    scriptContextPurpose :: BuiltinData
  }
  deriving (P.Eq)

-- PlutusTx.Eq instance, needed by cooked-validators
instance Eq TweagScriptContext where
  (==) = (==)

PlutusTx.unstableMakeIsData ''TweagScriptContext

-- * Main Tweag oracle validator function

-- | Takes a Tweag oracle datum, redeemer and script contexts and attempts to
-- validate the oracle UTXO consumption.
--
-- - If the redeemer is Reclaim, checks that the beneficiary has signed and the
-- deadline has passed.
--
-- - If the redeemer is Execute, checks that the deadline has not passed, the
-- correct Marlowe input is present in the transaction, and that the associated
-- Marlowe redeemer has the right shape.
{-# INLINEABLE tweagOracleValidator #-}
tweagOracleValidator :: TweagOracleDatum -> TweagOracleRedeemer -> TweagScriptContext -> Bool
tweagOracleValidator
  -- In the Reclaim case, we only rely on the beneficiary and deadline from the
  -- datum, and the txInfoSignatores and txInfoValidRange from the script
  -- context which is made possible by using NameFieldPuns. We could have
  -- avoided this language extension by writing, for instance,
  -- (TweagOracleDatum _ _ _ deadline beneficary)
  TweagOracleDatum {beneficiary, deadline}
  Reclaim
  (TweagScriptContext TweagTxInfo {txInfoSignatories, txInfoValidRange} _) =
    (&&)
      (traceIfFalse "The transaction wasn't signed by the beneficiary" beneficiaryAmongSigners)
      (traceIfFalse "The deadline hasn't passed yet" afterDeadline)
    where
      -- This is the equivalent of Api.txSignedBy applied on the whole txInfo,
      -- which does not make much sense as only the signatories are necessary.
      beneficiaryAmongSigners = beneficiary `elem` txInfoSignatories
      afterDeadline = Api.from (deadline + 1) `Api.contains` txInfoValidRange
tweagOracleValidator
  -- In the Execute case, we need different parts of the Datum and script
  -- context, for which we use NameFieldPuns as well, thus allowing to directly
  -- see which parts are use, and which are not (as opposed to RecordWildCards)
  TweagOracleDatum {choiceName, marloweUtxo, deadline}
  Execute
  (TweagScriptContext TweagTxInfo {txInfoValidRange, txInfoRedeemers} _) =
    (&&)
      (traceIfFalse "The deadline has passed" beforeDeadline)
      -- We either traces the error message if any, or compare the embedded
      -- name with the expected one, tracing another error if they differ
      ( case eitherErrorOrRedeemerChoiceName of
          Left err -> trace err False
```

```
193             Right redeemerChoiceName -> traceIfFalse "The choice names don't match" $ redeemerChoiceName == choiceName
194         )
195       where
196         beforeDeadline = Api.to deadline `Api.contains` txInfoValidRange
197         -- This lives in the Either monad. We attempt to retrieve the name
198         -- embedded in the Marlowe redeemer, while specifying relevant error
199         -- messages along the way when necessary.
200         eitherErrorOrRedeemerChoiceName = do
201           Api.Redeemer marloweRedeemerData <-
202             maybeToEither "The Marlowe input is not spent with a redeemer in the transaction"
203               $ PlutusTx.lookup (Api.Spending marloweUtxo) txInfoRedeemers
204           marloweRedeemer <-
205             maybeToEither "The Marlowe redeemer is not a list of inputs"
206               $ Api.fromBuiltinData @[Input] marloweRedeemerData
207           case marloweRedeemer of
208             [NormalInput (IChoice (ChoiceId redeemerChoiceName _) _)] -> return redeemerChoiceName
209             [MerkleizedInput (IChoice (ChoiceId redeemerChoiceName _) _) _ _] -> return redeemerChoiceName
210             _ -> Left "The Marlowe redeemer is not a singleton"
211
212 -- | Helper function to convert a Maybe into a Either with a given error
213 {-# INLINEABLE maybeToEither #-}
214 maybeToEither :: err -> Maybe a -> Either err a
215 maybeToEither e Nothing = Left e
216 maybeToEither _ (Just v) = Right v
217
218 -- | Untyped Tweag oracle validator, deserializing the various BuiltinData to
219 -- the right types and calling the main validator function.
220 {-# INLINEABLE untypedTweagOracleValidator #-}
221 untypedTweagOracleValidator :: BuiltinData -> BuiltinData -> BuiltinData -> ()
222 untypedTweagOracleValidator dat red ctx =
223   case do
224     typedDat <- Api.fromBuiltinData @TweagOracleDatum dat
225     typedRed <- Api.fromBuiltinData @TweagOracleRedeemer red
226     typedCtx <- Api.fromBuiltinData @TweagScriptContext ctx
227     return $ tweagOracleValidator typedDat typedRed typedCtx of
228     Nothing -> traceError "Error during deserialization"
229     Just b -> check b
230
231 -- * TweagOracle typed validator
232
233 -- | Empty data type to tag the typed validator
234 data TweagOracle
235
236 -- | Typed validator datum and redeemer instances
237 instance Scripts.ValidatorTypes TweagOracle where
238   type DatumType TweagOracle = TweagOracleDatum
239   type RedeemerType TweagOracle = TweagOracleRedeemer
240
241 -- | Typed Tweag oracle validator
242 typedTweagOracleValidator :: Script.TypedValidator TweagOracle
243 typedTweagOracleValidator =
244   C.validatorToTypedValidator
245     $ Script.mkValidatorScript $$(PlutusTx.compile [||untypedTweagOracleValidator||])
```

A.3 Metrics

We have performed a comparison between Wofram's and Tweag's oracle validators to emphasize the benefits of our various suggestions and optimizations.

A.3.1 Script size

We have written a function `validatorByteSize` which outputs the size in bytes of a typed validator. Here are the results:

```
ghci> validatorByteSize oracleFinalContractInst
10697
ghci> validatorByteSize typedTweagOracleValidator
7320
```

There is a decrease of 32% of the script size. This is likely mostly due to the remediation of issue 2.4.2.3.

A.3.2 Script resource consumption

When running traces with `cooked-validators` and activating the flag `pcOptPrintLog = True` in the pretty printer option, we can see the fees that have been computed for each transaction. These fee are directly correlated to the script resource consumption in terms of memory and cycles. Here is the table of the computed fee for both the *Execute* and *Reclaim* transaction in Wolfram's and Tweag's oracle validators.

	Execute transaction fee	*Reclaim* transaction fee
Wolfram Oracle	670797 lovelace	662085 lovelace
Tweag Oracle	510417 lovelace	501705 lovelace
Gain	23.9%	24.2%

We can see that there is a net gain of almost 25% of the transaction cost. This is likely mostly due to the remediation of issue 2.4.3.1.