# DEX Uniswap v2
# Sophia contracts on the Aeternity chain

# Audit Report

Test Object:
**DEX Uniswap v2 contracts**

Date: **April 19, 2022**

*This report presents the results of an audit performed for **DEX Uniswap v2 smart contracts**. The Sophia code for these contracts has been carefully reviewed, tested and analyzed for security vulnerabilities.*

*The review of the code was performed in two stages from mid January 2022 until March 24, 2022. In the first stage Quviq suggested improvements and slight modifications to make the resulting contract more efficient, more secure and easier to use.*

*The goal of this code review has been to make sure that the Uniswap v2 implemented is first and foremost secure, but also that it is efficient and easy to use. Quviq did not find any critical issues in the contracts, however there were a number of smaller issues with the code regarding documentation, readability and efficiency. In the final contract all issues discussed in this report have been addressed.*

*We see that it was highly efficient to do the review in multiple stages - giving both the reviewers and the developers time to discuss, evaluate and adapt. We have been able to address several issues in an efficient manner using this approach.*

# Introduction

The reviewed contracts is a Uniswap v2 implementation. The implementation consists of a factory contract, a token pair swap contract and high-level, more user friendly, router contract. The reviewed repository also contains a token contract for "wrapped aeternity tokens". The implementation closely follow https://github.com/Uniswap/v2-core and https://github.com/Uniswap/uniswap-v2-periphery respectively.

## Test object

The final Uniswap v2 contracts we have reviewed are retrieved from the master branch of the following Github repository with indicated SHA commit hash around April 1, 2022: **`https://github.com/aeternity/dex-contracts-v2/53d86c758e8470f051a64710413a01afc03a8708`**

Note that the review is limited to the contract code. Surrounding (interface) code and supplied tests have not been reviewed.

The review initially started at commit hash **`665d6c0236403ce78cb3200cf8c9e4f8537b4a28`** however, we quickly discovered a problematic issue with allowance in the wrapped aeternity token contract - this was rapidly fixed and the review was then done for commit hash **`cd7e6e550fd1e51f756890647cee8bf8b1a0ed0a`**

We only focused on functional correctness and potential security issues related to the contract. Possible user interfaces interacting with the contract are not part of this review.

## Deployment assumptions

We have assumed that the contract will be compiled with the Sophia compiler [3] version 6 for FATE VM version 2. We expected the contract to run on Aeternity protocol Iris.

## Methodology

We reviewed the contracts using two complementary techniques. Model based testing and manual code inspection.

Model based testing builds upon automatic generation of a large set of tests. The generation approach helps finding unforeseen software errors. These delicate software errors are often caused by a sequence of events that bring the system in an unexpected state. A tester will normally try to write complex tests that bring the system in such states, steered by the (limited) imagination of what may happen. In model testing we let the computer explore possible paths that may happen and generate tests in order to try to provoke such paths.

We made a state based test model using QuickCheck [1, 2]. In short this will allow us to generate wellformed test cases that we can run against the reviewed contract. The model specifies what the tested smart contract entrypoint call does, and allows us to chain together sequences of calls with a known expected outcome.
This is a powerful test technique and it gives us good confidence that the reviewed contract behaves as expected in many different scenarios.

For smart contracts, we typically generate tests for sequences of possible contract entrypoints. In this sense, the **`init`** entrypoint is also part of the test case and defines the initial configuration of the contract(s). For Uniswap v2, we mainly tested interfacing the token swaps through the router contracts. After initialization (this includes creation of accounts, a token factory, a dummy token pair contract to clone, etc), the tests generated have arbitrary sequences of `add_liquidity`, `remove_liquidity` and of course `swap`. While doing these calls the model keeps track of (wrapped) tokens and liquidity and make sure that it all sums up at the end of the test.

Thousands of tests were generated and ran against the contracts, giving us good assurance that the contracts function correctly.

We also performed a manual code inspection with several pairs of eyes. Here we looked at each entrypoint in detail, checking for problematic inputs and unwanted behavior. We also looked for inefficient and/or unclear code, that makes it harder to understand, and possibly more expensive to run.

Testing and manual code inspection cannot guarantee to find all possible problems and security vulnerabilities, as mentioned in the Disclaimer, but by following a systematic approach we have made our review as thorough as possible in the given amount of time.

## Severity classification

We use the following classification scale for the issues we encounter.

- P1: Highest priority; loss of funds, deadlock of contract, hijack control, minting of tokens.
- P2: Severe disruption; No funds technically lost, DDOS, unusable protocol for extended period.
- P3: Inconvenient, or unexpected behavior of the protocol; loss of fund due to user error, easy to make mistakes, excessive gas usage.
- P4: Code organization, performance, clarity, gas consumption.
- P5: Cosmetic / Documentation.

## Disclaimer

In this report Quviq presents an informational exercise documenting the due diligence involved in the secure development of the Sophia smart contract, and make no material claims or guarantees concerning the contract's operation post-deployment. Under no circumstances Quviq can be liable for any direct or indirect consequences arising out of the deployment or use of this smart contract.
This report makes no claims that its analysis is fully comprehensive.There always is a possibility of human error in the review process or a contextual change in the area in which the contract operates. We recommend always seeking multiple opinions and audits.

This report does not constitute legal or investment advice.

# List of findings

## F01 - Incorrect allowance change logic in WAE

--- Severity: P3

--- Round: Initial check

The checks for allowance change in the wrapped æternity token contract (`contracts/WAE.aes`) are incorrect. This makes it impossible to change the allowance to a larger value - i.e. once you reach 0 you are stuck.

--- Recommendation

Adjust the logic to allow users to correctly set the allowance.

--- Status: **Fixed immediately, before the main audit.**

## F02 - Inefficient factory state organization

--- Severity: P3

--- Round: Main review

In AedexV2Factory (`contracts/AedexV2Factory.aes`) the type used for the record of all swapping pairs is

```
map(IAEX9Minimal, map(IAEX9Minial, IAedexV2Pair))
```

it would be better (more efficient and less error-prone) to use the type

```
map({IAEX9Minimal,IAEX9Minimal}, IAedexV2Pair)
```

--- Recommendation

Change the type (to a map indexed by the pair of token addresses) and adapt (simplify) the code interfacing it.

--- Status: **Fixed in the final version.**

## F03 - Bad naming in AedexV2Router

--- Severity: P4

--- Round: Main review

In AedexV2Router (`contracts/router/AedexV2Router.aes`) several of the helper functions are poorly named. For example `add_liquidity'` - a pure function that computes optimal liquidity values.

--- Recommendation

Names should reflect what the function does - this makes it easier for the reader to understand what is going on.

--- Status: **Fixed in the final version.**

## F04 - Duplicated check in AedexV2Router

--- Severity: P4

--- Round: Main review

In AedexV2Router (`contracts/router/AedexV2Router.aes`) the check

```
require(amount_a_optimal =< amount_a_desired,
"AedexV2Router: OPTIMAL_GREATER_THEN_DESIRED")
```

Will never fail - it is mathematically impossible given how these values are computed.

--- Recommendation

Remove the unnecessary check.

--- Status: **Fixed in the final version.**

## F05 - Possible simplification of burn_update in AedexV2Pair

--- Severity: P4

--- Round: Main review

In AedexV2Pair (`contracts/AedexV2Pair.aes`) the function `burn_update` is always called for `Contract.address` and the `value` is always equal to the balance of `Contract.address` - thus it would be clearer to remove the arguments and possibly rename the function.

--- Recommendation

Simplify the function - it will read better and it will be less error-prone if someone decided to change it later.

--- Status: **Adapted in the final version.**

## F06 - Inefficient/hard to read state update in WAE

--- Severity: P4

--- Round: Main review

In the WAE contract (`contracts/WAE.aes`) the state updates are unnecessarily complex. For example

```
put(state{balance_of @ b = b {[Call.caller = 0] @ value = value + Call.value}})
```

would be better (more efficient and easier to read) written as

```
put(state{balance_of[Call.caller = 0] @ val = val + Call.value})
```

--- Recommendation

Change the state updates to the more efficient format, making use of the syntactic sugar of Sophia.

--- Status: **Fixed in the final version.**

## F07 - Underdeveloped documentation

--- Severity: P5

--- Round: Main review

There is a distinct lack of documentation - this comes from the implementation being more or less a clone of a similar implementation in Solidity. However, the repository should be self-contained and it should be possible to read the code without having to also read the Solidity code. This is especially true for entrypoint functions that should be well documented.

Further, minimum liquidity is an important concept - when it is set there should be a comment of what a good value is.

Another example, the debug mode is a bit problematic; The Factory has a good warning, it would make sense to extend the documentation of this a bit - stressing the fact that with a correctly setup Factory all the Pool/Pairs will also be correct.

There are a lot of contracts/interfaces in the `contracts` directory, a top-level description of what is used where would help.

In AedexV2Router; the feature that the pair is created if it doesn't exist (when adding liquidity) is somewhat hidden - more documentation will help.

--- Recommendation

Add and/or extend the documentation to a level that can be expected of a project this size and complexity.

--- Status: **Massively improved in the final version, still a bit thin at the top level/ project overview.**

# Recommendation

Most of the earlier recommendations are followed and there are no new issues found in the final review, **we have no more recommendations**. We still believe the documentation can further improved - but that process lies outside the scope of this review.

# References

[1] Arts, T., Hughes, J.: How well are your requirements tested? In: Briand, L., Khurshid, S. (eds.) International Conference on Software Testing, Verification and Validation (ICST). IEEE (April 2016)

[2] Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing Telecoms Software with Quviq QuickCheck. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang. pp. 2–10. ERLANG '06, ACM, New York, NY, USA (2006)

[3] Ulf Norell, Hans Svensson, et. al., Sophia Compiler, https://github.com/aeternity/aesophia