CavalRe AMM

Smart Contract Security Assessment

Sep. 10, 2022

ABSTRACT

Dedaub was commissioned to audit the CavalRe protocol implementation, at https://github.com/CavalRe/amm. The protocol implements a novel automated market maker (AMM) design, based on the principles of the technical paper "A Family of Multi-Asset Automated Market Makers". The result is an AMM that supports seamlessly multiple-pool assets, as well as unifies staking/unstaking and swapping, since the pool liquidity token is just another asset in the pool, with a negative weight (i.e., counted as a "liability").

SETTING AND CAVEATS

The audit report covers commit hash 63d8e19ad14da0a1cc28dd509cdb48c3d4dcc2c9. Two auditors and a mathematician worked on the codebase over 4 days. The scope of the audit is limited to a single file (Pool.sol).

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e., issues in "regular use") is a secondary consideration. Functional correctness of most aspects (e.g., relative to low-level calculations, including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing. Importantly, thorough integration testing in the setting of final use is also an aspect that is not effectively covered by human auditing and remains the responsibility of the development team.

Economic considerations are unavoidable when attempting to understand the protocol in depth. However, the protocol creators are explicitly "*not expecting [Dedaub] to audit the economics. Just the contract security.*" We offered informal advisory comments on the financial model (esp., which elements were unclear) in a separate document.

PROTOCOL-LEVEL CONSIDERATIONS

The main threats for the protocol will be economic threats. The model described in the technical paper admits an infinite number of solutions, and the code currently implements just one of them, materializing many design choices. The design choices should be assessed with thorough studies, through simulation, and with clear argumentation for why the protocol behaves desirably. In addition to standard correctness of regular trades, one should be concerned about the possibility of high price impact (colloquially, "slippage"), the impact of fees on weight adjustment, and more.

Specifically, the code implements a specialization of the model in the technical paper for the parameter $k = 1$. However, many more design decisions are implicitly encoded. These design decisions should be made explicit in a whitepaper, with careful consideration of their impact and validation that the final AMM behaves desirably (possibly in extensive simulations). We did not have access to the full test suite, which is currently under development. However, we note that a good test suite for a protocol of this kind should do more than achieve code coverage, multiply covering both usual and extreme scenarios, with explicit automatic checks of financial soundness: numbers should be cross-checked automatically against expected or admissible behavior; scenarios should be tested against others that should behave equivalently; confluent behaviors (i.e., multiple ways to get to the same final state) should be exercised and cross-checked.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contracts. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description

CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: -User or system funds can be lost when third party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to programming error.
LOW	Examples: -Breaking important system invariants, but without apparent consequences. -Buggy functionality for trusted users where a workaround exists. -Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	Duplicate tokens in the `payTokens` or `receiveTokens` arrays of `multiswap` cause surprising behavior	**OPEN**

If the caller supplies the same token multiple times in the payTokens or receiveTokens parameters of multiswap, the behavior is *not* equivalent to supplying the token once, with an amount equal to the sum of the previous amounts.

For instance, the computation of fracValueIn would have a different denominator in the case of splitting the token amounts:

```
    // Compute fracValueIn
    {
      for (uint256 i; i < payTokens.length; i++) {
          address payToken = payTokens[i];
          uint256 amountIn = amounts[i];
          if (payToken == address(this)) {
             _balance -= amountIn;
             fracValueIn += amountIn.ddiv(_balance);
          } else {
             Asset storage assetIn = _assets[payToken];
             uint256 gamma = DMath.ONE - assetIn.fee;
             uint256 weightIn = assetIn.scale.ddiv(_scale);
             fracValueIn += gamma.dmul(weightIn).dmul(amountIn).ddiv(
                      assetIn.balance + amountIn
             );
             assetIn.balance += amountIn;
```

```
            }
        }
    }
```

We recommend preventing duplicates inside these arrays, just as duplicates are prevented between them.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	The `uninitialize()` function leaves contract in an unusable state.	**OPEN**

The `uninitialize()` function allows the contract owner to withdraw any tokens deposited through the `addAsset()` function, in case of an erroneous initialization of the contract. Although funds can be retrieved through this method, the state of the `_addresses` array and `_assets` mapping is not reversed, making the contract unusable.

LOW SEVERITY:

ID	Description	STATUS
L1	The `addAsset` function overwrites previous information	**OPEN**

Calling the `addAsset` function will overwrite previous information for the same asset, and generally leaves the contract in an inconsistent state in case of a duplicated asset.

```
function addAsset(
```

```
        address payToken_,
        uint256 balance_,
        uint256 fee_,
        uint256 assetScale_
) public nonReentrant onlyOwner {
  ...
  _assets[payToken_] = Asset(
            IERC20(payToken_),
            IERC20Metadata(payToken_).name(),
            IERC20Metadata(payToken_).symbol(),
            decimals_,
            balance_,
            fee_,
            assetScale_
  );
}
```

Since this is owner-only functionality, the behavior is not expected to be a concern. However, it would not be surprising for the owner to add the same asset twice (different amounts) and expect that the state would be updated additively. Additionally, this issue conspires with M1: once an asset is added with a wrong balance or other parameter, it is not easy to fix its allocation without completely invalidating the pool.

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Inefficient use of memory in `uninitialize()`.	**INFO**

In the function `uninitialize()`, the `asset` struct corresponding to a token address is copied in its entirety to memory, when only the balance and decimals fields need to be read, making the implementation gas-inefficient.

```
function uninitialize(address token) public nonReentrant onlyOwner {
      if (_isInitialized == 1) revert AlreadyInitialized();

      Asset memory x = _assets[token];
      SafeERC20.safeTransfer(
            x.token,
            owner(),
            toCanonical(x.balance, x.decimals)
      );
}
```

The `balance` and `decimals` fields can be read directly from the struct via a storage pointer, instead of a memory buffer, and the token address is already available as a parameter.

```
function uninitialize(address token) public nonReentrant onlyOwner {
      if (_isInitialized == 1) revert AlreadyInitialized();

        Asset storage x = _assets[token];
        SafeERC20.safeTransfer(
            IERC20(token),
            owner(),
            toCanonical(x.balance, x.decimals)
        );
    }
```

A2	Expensive operations within loops inside `multiswap()`	**INFO**

The multiswap operation performs a number of expensive writes into storage variables inside of loops. The resulting implementation is gas inefficient. Specifically, the two loops below can improve gas-performance-wise:

```
// Transfer tokens to the pool
for (uint256 i; i < payTokens.length; i++) {
…
      _assets[payToken].scale += delta;
      _scale += delta;
    }
}

// Transfer tokens to the receiving address
for (uint256 i; i < receiveTokens.length; i++) {
…
      _assets[receiveToken].scale += delta;
      _scale += delta;
    }
}
```

Instead of accumulating values directly into storage locations, one can instead accumulate into a local variable. The local variable can then be written once into `_assets[*],scale` and `_scale` at the end of each loop. (This fix would also address another current inefficiency, of computing `_assets[payToken]` and `_assets[receiveToken]` twice within each loop, resulting in extraneous SLOADs.)

A3	Redundant check in `stake()`	**INFO**

The `stake()` function performs a redundant check to determine whether the token field of the assetOut struct is the same as receiveToken. However this check will always yield false since `assetOut` must be the struct corresponding to that token address (or there is an internal error in the configuration of the pool).

```
Asset storage assetOut = _assets[receiveToken];
// Check if unstake
if (address(assetOut.token) != receiveToken)
```

```
revert InvalidUnstake(receiveToken);
```

A4	Name of `toCanonical` confusing	INFO

The `toCanonical` function should perhaps rather be called `fromCanonical` since it translates from a canonical precision, e18, to the decimals the token expects.

A5	Magic constants are best avoided	INFO

Constants such as 10000, 3, 4, which appear inlined in the code, are best avoided, in favor of named constants that explain the intent and can be changed uniformly everywhere with a single, local edit.

```
        if (amounts[i] < 10000) revert TooSmall(amounts[i]);
```

A6	The order of operations may lose precision	INFO

In `swap`, performing the ddiv operation before multiplications may lose precision. This is likely not a concern, since 18 digits are being kept, but we also see no danger of overflow for realistic protocol quantities. (Correctness should be ascertained by testing with extreme values after implementing this suggestion.)

```
        uint256 invGrowthOut = DMath.ONE +
              gamma.dmul(weightRatio).dmul(amountIn.ddiv(reserveIn));
```

Similarly in multiswap:

```
factor = gamma.ddiv(weightOut).dmul(allocation).dmul(fracValueIn)
```

A7	Improve loose condition	INFO

In `unstake`, there is a condition restricting `amountIn` (the amount of pool tokens a user can burn and get back his stake) to a maximum value of `4/3*_balance`. But, since this amount is subtracted from `_balance`, the maximum allowed value for `amounIn` should be equal to `_balance` or less.		
A8	Compiler bugs	**INFO**
The code is compiled with Solidity 0.8.16. Version 0.8.16, in particular, has some known bugs, which we do not believe to affect the correctness of the contracts.		

CENTRALIZATION ASPECTS

The protocol is highly decentralized, with all owner privileges relinquished after pool deployment.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.