CavalRe AMM

Smart Contract Security Assessment

July 10, 2023

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, which supports swaps and multiswaps, adding and removing liquidity as well as staking and unstaking.

SETTING AND CAVEATS

The audit report covers commit hash 5db05fa6eb281c1736fdbea70ac17a1e6b41a017.

The audit scope consists of the following files:

```
contracts/
├── LPToken.sol
├── 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.

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	Inheritance issue leads to operations being performed twice	**RESOLVED (4697d3cd)**
The LPToken contract inherits from the ERC20 contract. It then overrides both its external functions such as `transfer`, as well as its internal functions such as `_transfer`. Both the overridden external and internal functions adjust the amounts they receive before calling the corresponding version in the super contract. Now, if a user invokes an external overridden function such as `LPToken::transfer`, the amount is adjusted and then `ERC20::transfer` is called. Then `ERC20::transfer` will try to call `ERC20::_transfer`, but seeing this has been overridden, will call `LPToken::_transfer` instead, where the amount is adjusted again. Finally `LPToken::_transfer` will call `ERC20::_transfer` to finalise the transfer operation. As a result, the amount has been adjusted twice instead of once. `LPToken::approve` and `LPToken::_approve` also suffer from the same issue.		
H2	`Pool::Multiswap` allows LPToken to appear on both sides of a multiswap	**RESOLVED (4697d3c)**
An error in the validation of `Pool::multiswap` fails to ensure that the LPToken only appears on one side of a multiswap.		

The error occurs because the LPToken, unlike other tokens, is not tracked in the check_ array, and the isLP variable, which tracks that it has been seen among the payTokens, is reset before the receiveTokens are checked.

Pool::multiswap

```
function multiswap(
        address[] memory payTokens,
        uint256[] memory amounts,
        address[] memory receiveTokens,
        uint256[] memory allocations
    )
        public
        nonReentrant
        onlyInitialized
        onlyAllowed
        returns (uint256[] memory receiveAmounts)
    {
        ...

        // Check duplicates
        {
            bool isLP;
            uint256 temp;
            bool[] memory check_ = new bool[](_assetAddress.length);
            for (uint256 i; i < payTokens.length; i++) {
                address token = payTokens[i];
                if (token == address(0)) revert ZeroAddress();
                if (address(this) == token) {
                    if (isLP) revert DuplicateToken(token);
                    isLP = true;
                    if (i != 0) {
                        payTokens[i] = payTokens[0];
                        payTokens[0] = address(this);
                        temp = amounts[i];
                        amounts[i] = amounts[0];
                        amounts[0] = temp;
```

```
                }

                //Dedaub: LPToken not added to check_
                //        because loop breaks early

                continue;
            }
            AssetState memory asset_ = _assetState[token];
            if (asset_.token != token) revert AssetNotFound(token);
            if (check_[asset_.index]) revert DuplicateToken(token);
            check_[asset_.index] = true;
        }

        //Dedaub: presence of LP token is erased

        isLP = false;
        for (uint256 i; i < receiveTokens.length; i++) {
            address token = receiveTokens[i];
            if (token == address(0)) revert ZeroAddress();
            if (address(this) == token) {
                if (isLP) revert DuplicateToken(token);
                isLP = true;
                if (i != 0) {
                    receiveTokens[i] = receiveTokens[0];
                    receiveTokens[0] = address(this);
                    temp = allocations[i];
                    allocations[i] = allocations[0];
                    allocations[0] = temp;
                }
                continue;
            }
            AssetState memory asset_ = _assetState[token];
            if (asset_.token != token) revert AssetNotFound(token);
            if (check_[asset_.index]) revert DuplicateToken(token);
            check_[asset_.index] = true;
        }
```

```
        }
        ...
}
```

H3	Erroneous geometric mean updates	**RESOLVED (5db05fa)**

The functions `_increaseBalance` and `_decreaseBalance` of the Pool.sol contract updates the balance variable of `_poolState` structure and also the meanBalance, calling the `_geometricMean` function. `_geometricMean` computes the new geometric mean of the balance using the time elapsed (delta) since the last update. The problem is that `_increaseBalance/_decreaseBalance` has already updated the `_poolState.lastUpdated` before calling `_geometricMean`, therefore delta will be zero and the geometric mean will be not updated and always stay the same.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	`Pool::removeAsset` not cleaning up properly	**RESOLVED (253a2a0)**

The `Pool::removeAsset` function does not delete the `_assetState[token]`, but just sets some of the asset variables to 0. Other fields, such as `_assetState[index]` and `_assetState[token]` do not change. This can have a number of consequences. For instance, if an asset is removed, it cannot be added again later, because the token field is already set. Also, swap and other functions will still work with the removed asset and transactions will only fail when a division by zero error is encountered due to a relevant field being set to 0.

Pool::removeAsset

```
function removeAsset(
        address token
    ) public nonReentrant onlyUninitialized onlyOwner {
        ...

        \\Dedaub: Only a subset of fields of the Asset struct are zeroed
        asset_.balance = 0;
        asset_.meanBalance = 0;
        asset_.scale = 0;
        asset_.meanScale = 0;
        asset_.lastUpdated = 0;

        ...
}
```

Pool::addAsset

```
function addAsset(
        address payToken_,
        uint256 balance_,
        uint256 fee_,
        uint256 assetScale_
    ) public nonReentrant onlyUninitialized onlyOwner {
        if (payToken_ == address(0)) revert ZeroAddress();

        \\Dedaub: Adding an asset after removing it will cause a revert
        if (_assetState[payToken_].token == payToken_)
            revert DuplicateToken(payToken_);
        ...
    }
```

LOW SEVERITY:

ID	Description	STATUS
L1	The fee model is gameable	**PARTIALLY RESOLVED (0eddc89)**
The fee model is not linear i.e. if a user splits a multiswap into many smaller ones, the total fees he will have to pay are not the same. Someone could take advantage of this and try to figure out optimal ways of arranging his (multi)swaps to minimize the fees he will pay.		

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	`LPToken::setProtocolFee` can be changed by owner	**INFO**
`LPToken::setProtocolFee` can be used by the owner to change the protocol fee without warning. The protocol fee is the proportion of the fees which goes to the protocol as opposed to the liquidity providers. The fee charged to the users cannot be		

changed by the owner. It is understood that liquidity providers may withdraw their liquidity if the protocol fee changes in an unacceptable way and that it is in the interest of the owner not to make abrupt changes to the protocol fee.

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	`LPToken::addUser` can also update users	**INFO**

The `LPToken::addUser` function can also be used to update a user, and not just to add a user. We advise to separate these two functionalities for clarity.

LPToken::addUser

```
function addUser(
        address user,
        uint256 discount
    ) public nonReentrant onlyOwner {
        if (user == address(0)) revert InvalidUser(user);
        UserState memory state = _userState[user];
        if (state.user != user) {
            _userAddress.push(user);
        }
        //Dedaub: User state is still updated if
        //        user exists

        _userState[user] = UserState(user, true, discount);
    }
```

A2	Spelling Mistake in Error parameter	INFO
The Pool contract has an error called InvalidSwap which has a parameter called receiveToke instead of receiveToken.		
A3	No minAmountOut for `Pool::swap` and `Pool::multiswap`	INFO
We recommend adding a minAmountOut parameter to `Pool::swap` and `Pool::multiswap`, so as to enable users to specify the minimum amount of received tokens they consider acceptable when invoking these operations.		
A4	`Pool::swap` and `Pool::multiswap` have a limit on the size of the payTokens but not on the size of the receiveTokens	INFO
`Pool::multiswap` and `Pool:swap` stop payTokens from amounting to more than 1/3 of the asset in the pool, but there is no corresponding requirement for receiveTokens. It is recommended to have a similar requirement for receiveTokens so as to avoid swaps and multiswaps from draining a particular asset of the pool.		
A5	Redundant updates of assets' indices	INFO
In `Pool::removeAsset` after the removal of the asset, the indices of all assets are updated. But only one index has actually changed, the index of the last asset in _assetAdress[], and all the other assets are reassigned their previous indices.		
A6	Unnecessary `nonReentrant` modifiers	INFO
Many functions of the LPToken.sol contract e.g. `setProtocolFee, setProtocolFeeRecipient, addUser,...` have a `nonRentrant` modifier, although none of them makes external calls (and moreover they are only callable by the owner of the contract).		

A7	Duplicate code	INFO
In Pool::multiswap there are two identical for-loops, checking if there are token repetitions in the set of `payTokens` and `receiveTokens`. The code could be refactored using a single method that will check for repetitions.		

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.