Muffin Protocol

Smart Contract Security Assessment

Feb 17, 2022

BACKGROUND

Dedaub was commissioned to perform an audit of the Muffin protocol. Two auditors worked on the code over the course of 3.5 weeks, and Dedaub's financial math consultant participated in significant parts of the audit.

Muffin is a concentrated liquidity protocol, with logic resembling that of Uniswap v3. It introduces significant innovation compared to Uniswap v3 and a full re-implementation of all functionality. Innovative elements (as captured in the Muffin technical documentation) include:

- A swap can use multiple fee tiers at once, in a near-optimal way. E.g., part of the amount swapped may be exchanged using a 0.3% fee pool's liquidity, while a different part is exchanged using a lower fee pool.
- Limit orders are supported, so that liquidity automatically stays in the form of one token (i.e., is withdrawn from the swap pool) once a desired price is reached.
- There is a single contract holding all pools, resulting in a more efficient implementation.
- The time-weighted average price (TWAP) oracle computes the geometric mean over the price in all fee tiers, and maintains exponential moving averages.
- Internal accounts for users of the protocol are maintained.

The codebase was audited at commit hash e5099f82bfe1c758ca3eb2163c91feaa4313b0ec. The contracts audited (excluding interface, mock, test functionality) are listed below:

```
MuffinHubBase.sol
MuffinHubPositions.sol
MuffinHub.sol
libraries/Constants.sol
libraries/Pools.sol
libraries/Positions.sol
libraries/Settlement.sol
libraries/TickMaps.sol
```

```
libraries/Ticks.sol
libraries/Tiers.sol
periphery/Manager.sol
libraries/math/EMAMath.sol
libraries/math/FullMath.sol
libraries/math/Math.sol
libraries/math/PoolMath.sol
libraries/math/SwapMath.sol
libraries/math/TickMath.sol
libraries/utils/PathLib.sol
libraries/utils/SafeTransferLib.sol
periphery/base/ERC721Extended.sol
periphery/base/ERC721.sol
periphery/base/ManagerBase.sol
periphery/base/Multicall.sol
periphery/base/PositionManager.sol
periphery/base/SelfPermit.sol
periphery/base/SwapManager.sol
```

SETTING & CAVEATS

The audited codebase is of a medium size at about 4KLoC. 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. Typically it can only be covered if we are provided with unambiguous (i.e., full-detail) specifications of what is the expected, correct behavior. We did expend significant effort considering functional correctness issues such as a) scaling; b) overflow; c) overall match of documented mathematical operations to the code functionality; d) logic correctness in the code. However, the client should be aware that any such effort is partial: functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing or verification rather than human auditing.

ARCHITECTURE & HIGH-LEVEL RECOMMENDATIONS

The codebase is among the most mature and sophisticated in the DeFi space. There is evidence of excellent software development practices and professional quality.

The security architecture is similarly strong. There is care to avoid reentrancy attacks at all callback points, which is a particularly serious threat given the single-contract nature of the implementation. Locking per-asset and per-pool affords strong protection (with an issue for multi-address tokens identified later). Token transfer operations are carefully designed to charge only verified parties (typically the external caller of Muffin functionality, maintained throughout the protocol's inter-contract transaction flow).

The main recommendation for code understandability would be to include more detailed comments over the implementation techniques used to realize the whitepaper functionality. Concepts such as liquidityLower/Upper, nextTickAbove/Below, feeGrowthInside/Outside would have greatly benefited from a precise definition.

MATHEMATICAL APPROACH & PRESENTATIONS

In the course of understanding the financials of the project, we developed alternative formulations of the optimization problem and the greedy algorithm approximating the full solution.

A. Alternative presentation suggestion

The protocol is based on the solution of the following constrained optimization problem:

$$\text{minimize } f(\delta_1, \ldots, \delta_n) = \sum_{i=1}^{n} \frac{L_i^2}{x_i + \gamma_i \delta_i}$$

$$\text{subject to } \sum_{i=1}^{n} \delta_i = \Delta,\ \delta_1, \ldots, \delta_n \geq 0$$

The solution of this problem using Lagrange multipliers is very elegant, but relies on math advanced enough to alienate some of the audience. An elementary solution can be obtained as follows.

We first solve the equation of the constraint for one of the variables i.e.:

$$\delta_n = \Delta - \sum_{i=1}^{n-1} \delta_i$$

Then, we substitute this expression into the formula of `f`. We get a new optimization problem with one less variable:

$$\text{minimize } h(\delta_1, \ldots, \delta_{n-1}) = \sum_{i=1}^{n-1} \frac{L_i^2}{x_i + \gamma_i \delta_i} + \frac{L_n^2}{x_n + \gamma_n \left(\Delta - \sum_{i=1}^{n-1} \delta_i\right)}$$

$$\text{subject to } \delta_1, \ldots, \delta_{n-1}, \Delta - \sum_{i=1}^{n-1} \delta_i \geq 0$$

Now, a solution to this problem is more straightforward. We just have to solve the equation:

$$\frac{\partial h}{\partial \delta_i} = -\frac{L_i^2 \gamma_i}{(x_i + \gamma_i \delta_i)^2} + \frac{L_n^2 \gamma_n}{\left(x_n + \gamma_n \left(\Delta - \sum_{i=1}^{n-1}\right)\right)^2} = 0$$

for `i=1,...,n-1`. If, for simplicity, we call λ the second term above, we get the same set of equations, as the one with the Lagrange multiplier approach.
The solution is:

$$\delta_i^* = \frac{L_i}{\sqrt{\gamma_i}} \frac{\Delta + \sum_{j=1}^{n} \frac{x_j}{\gamma_i}}{\sum_{j=1}^{n} \frac{L_j}{\sqrt{\gamma_j}}} - \frac{x_i}{\gamma_i},\ i = 1, \ldots, n$$

B. Optimality of the solution

We prove that the solution's implementation in SwapMath.sol always gives the optimal results. We also suggest a modification, which is maybe faster in some cases. We think that the concept of convexity, which we discuss below, can be helpful also for the second part of the algorithm, where concentrated liquidity is taken into account. A careful investigation of this second part in light of the concepts we discuss here can probably lead to an accurate estimate of the divergence between the solution given by the protocol and the absolute optimal one.

The solution

$$\delta^* = (\delta_1^*, \ldots, \delta_n^*)$$

is a priori just a critical point of h. We have to prove that it is actually a minimum and also that the constraints

$$\delta_i^*, \ldots, \delta_n^* \geq 0$$

are satisfied.

In general, for an optimization problem of the form

$$\begin{aligned} &\text{minimize } h(\delta_1, \ldots, \delta_n) \\ &\text{subject to } \delta_1, \ldots \delta_n \geq 0 \end{aligned}$$

we first find all the critical points (partial derivatives=0); check if they satisfy the constraints and if they are actually (local) minima (using second derivatives); and then solve the corresponding minimization problem for each boundary (δ=0). Finally, we compare the value of h at each of these points and we find its global minimum.

In the case in hand everything is much simpler, because the function `f` we want to minimize is strictly *convex* as it is a sum of strictly convex functions.

Therefore, `h` is also strictly convex, as a restriction of a strictly convex function to a hyperplane and for strictly convex functions there is a unique global minimum.

Using this remark, we conclude that the solution δ*, if every coordinate is non-negative, is the unique solution of the optimization problem.

Convexity also helps us to handle the case when some solutions are negative.

We have that `h` is a convex function and we want to minimize it on a convex polytope `C`, defined by `n` hyperplanes (the hyperplanes $\delta_i = 0$). If $\delta^*_{i\_o} < 0$, then for every δ in `C` there exists a t_o in `[0,1]` such that:

$$t_o\delta^* + (1 - t_o)\delta = (\delta'_1, \ldots, \delta'_{n-1}) = \delta'$$

and the `i_o` coordinate of δ' equals zero. But `h` is convex and $h(\delta^*) < h(\delta)$:

$$h(\delta') \leq t_o h(\delta^*) + (1 - t_o)h(\delta) \leq h(\delta)$$

As a conclusion, we get that the minimum of h over C, which is unique, because h is strictly convex, must belong to a hyperplane $\delta_{i\_o}=0$, for an index `i_o` such that $\delta^*_{i\_o}<0$. Checking the gradient of h at each of these hyperplanes (because of the special form of f, as a sum of convex functions), we conclude that the minimum belongs to the intersection of all these hyperplanes.

This argument proves that the algorithm implemented in SwapMath.sol finds indeed the optimal solution. Alternatively, someone can compute all δ^*_i, set all the negatives equal to zero (and not only the first one), and then solve the optimization problem for the remaining δ_i. In general the solution for the new optimization problem contains possibly negative δ and we have to apply this procedure inductively. The worst case asymptotic complexity of this algorithm is $O(n^2)$, the same as the complexity of the algorithm used

currently. However, we speculate that it may converge more quickly. Using tests for several choices of the parameters involved in the problem, someone could compare the mean case complexity of the two implementations.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. 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 errors.
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”, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

ID	Description	STATUS
C1	Reentrancy for tokens with multiple addresses	**RESOLVED (commits 9c69af66, 28c474aa)**

The main mechanism for reentrancy protection is the locking of an affected asset (whose balance is typically checked before a callback to untrusted code). For instance, the code in function `deposit` is typical:

```
function deposit(
      address recipient,
      uint256 recipientAccRefId,
      address token,
      uint256 amount,
      bytes calldata data
) external {
      uint256 balanceBefore = getBalanceAndLock(token);
      IMuffinHubCallbacks(msg.sender).depositCallback(token,amount,data);
      checkBalanceAndUnlock(token, balanceBefore + amount);

      accounts[token][getAccHash(recipient, recipientAccRefId)] += amount;
       …
}
```

This approach, however, is vulnerable to reentrancy attacks when the token being locked has multiple addresses. (Such exotic tokens exist–see https://github.com/d-xo/weird-erc20#multiple-token-addresses. TUSD is an example, with >$1B valuation.)

An attacker can call `deposit` (and other vulnerable functions), receive a call back to actually send the deposited tokens to the MuffinHub, and before doing so call `deposit` again, on the second address of the token. Since the token has two addresses, the lock does not prevent the second call, the attacker can send funds to Muffin, and the funds will be counted twice, for both token balances. This attack is particularly insidious since the two addresses of the token could be participating in entirely different pools–e.g., one address could be used in the TUSD <> ETH pool, while another is used in a TUSD <> DAI pool.

HIGH SEVERITY:

[No high severity issues identified]

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Fields in ERC721Extended that cannot be set	**RESOLVED**

ERC721Extended offers no way to set `tokenDescriptorSetter` or `tokenDescriptor`. To set those, the requirement is `msg.sender == tokenDescriptorSetter`, but the constructor gives no initial value for `tokenDescriptorSetter`.

```
constructor(string memory name_, string memory symbol_)
ERC721(name_, symbol_) {
      nameHash = keccak256(bytes(name_));
}
…
function setTokenDescriptor(address descriptor) external {
      require(msg.sender == tokenDescriptorSetter);
      tokenDescriptor = descriptor;
}
```

```
function setTokenDescriptorSetter(address setter) external {
      require(msg.sender == tokenDescriptorSetter);
      tokenDescriptorSetter = setter;
}
```

LOW SEVERITY:

ID	Description	STATUS
L1	Error-prone dead code	**RESOLVED**

In ManagerBase.sol function payHub is supposed to handle the necessary payments to the MuffinHub contract due to swaps, position opening or updating.

```
function payHub(
      address token,
      address payer,
      uint256 amount
) internal {
      // Dedaub: user pays with ETH
      if (token == WETH9 && address(this).balance >= amount) {
        // pay with WETH9
        IWETH(WETH9).deposit{value: amount}(); // wrap only what is needed to pay
        IWETH(WETH9).transfer(hub, amount);
        // Dedaub: this case seems to be unused
      } else if (payer == address(this)) {
        // pay with tokens already in the contract
        SafeTransferLib.safeTransfer(token, hub, amount);
        // Dedaub: user pays with 'token'
      } else {
```

```
        // pull payment
        SafeTransferLib.safeTransferFrom(token, payer, hub, amount);
    }
}
```

A user interacting with the protocol is able to pay either with ETH or with an ERC20 token (first and third case as denoted in the snippet above). The case where the payer is a Manager- contract itself (second case in the above snippet) seems to be legacy code, as it is never executed given the current codebase. We consider this finding of low severity since we currently see no way to exploit it, but could be proven to be error-prone in any future updates given its sensitive functionality.

L2	Unclear token holding discipline, potential for error	**RESOLVED (commits 422b3d14, e065176d)**

The Manager contract should never hold either ETH or WETH between external transactions, or an attacker can drain it of these balances in multiple ways. One way is to indirectly invoke the payHub functionality, in the immediately preceding issue, through a public system action. More straightforwardly, the attacker can call unwrapWETH or refundETH, defined in ManagerBase:

```
function unwrapWETH(uint256 amountMinimum, address recipient)
external payable {
    uint256 balanceWETH = IWETH(WETH9).balanceOf(address(this));
    require(balanceWETH >= amountMinimum, "Insufficient WETH");

    if (balanceWETH > 0) {
        IWETH(WETH9).withdraw(balanceWETH);
        SafeTransferLib.safeTransferETH(recipient, balanceWETH);
    }
}

function refundETH() external payable {
```

```
        if (address(this).balance > 0)
                SafeTransferLib.safeTransferETH(msg.sender,
                                                address(this).balance);
}
```

Given the danger for user funds, such assumptions (of never holding ETH/WETH in this contract) should be clearly documented.

We further note that it is certainly possible for the Manager to receive WETH (or any token) by accident, and not by explicit transfer. For instance, the Muffin API is of the following form (shown here for one case of a swap, `SwapManager::exactInSingle`):

```
function exactInSingle(
        address tokenIn,
        address tokenOut,
        uint256 tierChoices,
        uint256 amountIn,
        uint256 amountOutMinimum,
        address recipient,
        bool fromAccount,
        bool toAccount,
        uint256 deadline
) external payable checkDeadline(deadline) returns (uint256 amountOut) {
        (, amountOut) = IMuffinHub(hub).swap(
                tokenIn,
                tokenOut,
                tierChoices,
                amountIn.toInt256(),
                toAccount ? address(this) : recipient,
                toAccount ? getAccRefId(recipient) : 0,
                fromAccount ? getAccRefId(msg.sender) : 0,
                fromAccount ? new bytes(0) : abi.encode(msg.sender)
        );
        require(amountOut >= amountOutMinimum, "TOO_LITTLE_RECEIVED");
}
```

However, the swap call ends up executing the code below in MuffinHub:

```
// Dedaub: called from "swap"
function _transferSwap(
      address tokenIn,
      address tokenOut,
      uint256 amountIn,
      uint256 amountOut,
      address recipient,
      uint256 recipientAccRefId,
      uint256 senderAccRefId,
      bytes memory data
) internal {
      …
      if (recipientAccRefId == 0) {
            SafeTransferLib.safeTransfer(tokenOut, recipient, amountOut);
      }
      …
```

This means that if the caller of `exactInSingle` passes a 0-address `recipient` yet also sets the `toAccount` flag, the money goes to the Manager contract (`address(this)` in the `SwapManager::exactInSingle` code, which becomes `recipient` in `MuffinHub::_transferSwap`), and it is effectively unrecoverable/burned, or, if it is WETH, becomes available for anyone to take via one of the earlier-discussed ways.

A clearer model of the movement of funds, as well as safeguards, will prevent such errors. There is no reason to set `this` as the recipient when the funds are expected to go into an internal account (flag `toAccount`) and not to an external Ethereum account. (Perhaps setting it to `msg.sender` will prevent some misuse, although we have not confirmed that this will work with the rest of the code.) Furthermore, the code could check that the `toAccount` flag is never used with a zero-address `recipient`, since zero is used to indicate "`toAccount` was false" in downstream code, such as the `MuffinHub::_transferSwap` above.

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	Library contract imported twice	**DISMISSED**

In contract `MuffinHub.sol` importing Pools library is redundant:

```
import "./libraries/Pools.sol"

contract MuffinHub is IMuffinHub, MuffinHubBase { ... }
```

since it is already imported in the `MuffinHubBase` parent class:

```
import "./libraries/Pools.sol";

abstract contract MuffinHubBase is IMuffinHubBase { ... }
```

ID	Description	STATUS
A2	Misleading variables naming	**DISMISSED**

In SwapMath.sol, functions `calcTierAmtsIn`, `calcTierAmtsOut`, variables

```
uint256 num; //   numerator of sqrt lambda (sum of UQ128)
uint256 denom; //  denominator of sqrt lambda (sum of UQ200 + amount)
```

are falsely named since `num` is actually the denominator and `denom` the numerator of the solution.

In function `_ceilMulDiv` in the same file, the name `denom` is used correctly.

A3	Fallback/proxy code not compatible with further Solidity calls	INFO

The fallback code of MuffinHub is used to proxy (via `delegatecall`) into MuffinHubPositions. This is a simple way to split the contract functionality into two, avoiding the Ethereum contract size restrictions. The code used to proxy is taken from an OpenZeppelin proxy implementation.

```
fallback() external {
      address _positionController = positionController;
      assembly {
            calldatacopy(0, 0, calldatasize())
            let result := delegatecall(gas(),
                        _positionController, 0, calldatasize(), 0, 0)
            returndatacopy(0, 0, returndatasize())
            switch result
            case 0 {
            revert(0, returndatasize())
            }
            default {
            return(0, returndatasize())
            }
      }
}
```

Out of an abundance of caution, we warn that this implementation explicitly assumes that, upon return from the `delegatecall`, no further Solidity code will run: the return data is written in memory position 0, overwriting the Solidity scratchpad memory. There is currently no issue with this practice in the Muffin code: the fallback function cannot be called in a context that returns to Solidity code. However, given that the code base uses MultiCall (elsewhere) developers should be warned of the danger of possibly adding MultiCall functionality to MuffinHub in the future: a call delegated through the fallback function will return and further calls will be issued. Without much further low-level exploration, it is not clear that the memory state of MultiCall logic can be indeed affected, and even less that a security issue can arise from such a combination of MultiCall and the above fallback function, but the possibility is sufficient for an advisory note.

A4	Assert statements are used to catch invalid inputs	INFO

The Solidity `assert` statement is best used to signal that a condition that should never arise (i.e., it represents a logical or code error) has occurred. In contrast, `require` is the standard way to check for invalid inputs. The codebase often fails via an `assert` and not a `require` for invalid inputs. The main example is operations in the Math library:

```
library Math {
  /// @dev Compute z = x + y, where z must be non-negative
  /// and fit in a 96-bit unsigned integer
  function addInt96(uint96 x, int96 y) internal pure returns (uint96 z) {
      unchecked {
            int256 s = int256(uint256(x)) + int256(y);
            assert(s >= 0 && s <= int256(uint256(type(uint96).max)));
            z = uint96(uint256(s));
      }
  }
  ...
  function toInt96(uint96 x) internal pure returns (int96 z) {
      assert(x <= uint96(type(int96).max));
      z = int96(x);
  }
```

The above function, `toInt96`, for instance, is used in one of the most major safeguards in the system: in checking that the liquidity parameter supplied to a `burn` operation (an unsigned integer) does not become negative when cast into a signed integer. (Having this would enable an attacker to burn a negative amount of liquidity, i.e., to add enormous liquidity at will.) Therefore, the `assert` checks external input and should be a `require`.

It is possible that the developers considered this and chose the `assert` in order to cause greater damage (by burning all supplied gas, as `assert` does) to a caller that supplies such invalid values. However, the practice is still questionable.

A5	Misleading comments	RESOLVED

In library `Ticks.sol` the NatSpec comments explaining a `Tick`'s parameters `needSettle0`, `needSettle1` refer to false limit order types:

```
* @param needSettle0    True if needed to settle positions with lower tick
boundary at this tick (i.e. 0 -> 1 limit orders)
* @param needSettle1    True if needed to settle positions with upper tick
boundary at this tick (i.e. 1 -> 0 limit orders)
```

Actually, `needSettle0` becomes true if a OneToZero limit order should be settled, while `needSettle1` if a ZeroToOne one.

In the Math library, the following comment is misleading:

```
/// @dev Compute z = max(x - y, 0) and r = x - z
/// Dedaub: above is false
function subUntilZero(uint256 x, uint256 y)
internal pure returns (uint256 z, uint256 r) {
      unchecked {
            if (x >= y) z = x - y;
            else r = y - x;
      }
}
```

A6	Multiple definitions of constants	**DISMISSED**

Some constants are defined in both TickMath and Constants.sol:

```
int24 internal constant MIN_TICK = -776363;
int24 internal constant MAX_TICK = 776363;
uint128 internal constant MIN_SQRT_P = 65539;
uint128 internal constant MAX_SQRT_P =
  340271175397327323250730767849398346765;
```

A7	Magic constant	RESOLVED

In `Pools::_addTier`, `Pools::setTierParameters`, 100000 is a "magic constant" and should ideally be given a name.

```
function _addTier(
      Pool storage pool,
      uint24 sqrtGamma,
      uint128 sqrtPrice
) internal returns (uint256 amount0, uint256 amount1) {
      uint256 tierId = pool.tiers.length;
      require(tierId < MAX_TIERS);
      require(sqrtGamma <= 100000);
      …
```

A8	Refactor common code?	RESOLVED

In `Positions::update`, it might be more elegant to factor out the statement

`self.liquidityD8 = liquidityD8New`

which is common in all three branches (and a no-op in the unlikely case that no branch is taken).

```
   function update(
        Position storage self,
        int96 liquidityDeltaD8,
        uint80 feeGrowthInside0,
        uint80 feeGrowthInside1,
        bool collectAllFees
    ) internal returns (uint256 feeAmtOut0, uint256 feeAmtOut1) {
        unchecked {
            ...
            if (collectAllFees) {
                ...
                self.liquidityD8 = liquidityD8New;
            } else if (liquidityDeltaD8 > 0) {
                ...
```

```
            self.liquidityD8 = liquidityD8New;
            ...
        } else if (liquidityDeltaD8 < 0) {
            ...
            self.liquidityD8 = liquidityD8New;
        }
    }
}
```

A9	Compiler known issues	INFO

The contracts were compiled with the Solidity compiler v0.8.10 which, at the time of writing, doesn't have any known bugs.

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, as well as a public bug bounty program.

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.