OneRing Audit

Smart Contract Security Assessment

July 21, 2022

ABSTRACT

Dedaub was commissioned to perform a security audit of the OneRing protocol.

This covered OneRing's contracts at commit hash:
`59a7a11f9048d3c78410a3e0d30fa782a017df03` of the *master* branch.

The audited contract list is the following:

- `contracts/MasterChefStrategy.sol`
- `contracts/facets/ManagementFacet.sol`
- `contracts/facets/PublicInfoFacet.sol`
- `contracts/facets/VaultFacet.sol`
- `contracts/facets/OwnershipFacet.sol`
- `contracts/libraries/LibManagement.sol`
- `contracts/libraries/LibVault.sol`
- `contracts/libraries/LibUtils.sol`
- `contracts/libraries/LibStorage.sol`
- `contracts/libraries/LibRequirements.sol`
- `contracts/libraries/LibMasterChefUtils.sol`

Two auditors worked on the codebase for a week.

As a result of the audit, the OneRingTeam fixed a number of issues. These fixes were also reviewed by the auditors. These are located at commit hash:
`a68171dac9b300f4f9f91e3eb10f5a5abb69be8a`

Setting and Caveats

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. In terms of functional correctness, we often trusted the code’s calculations and interactions, in the absence of any other specification. Functional correctness 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. The scope of the audit includes smart contract code. Interactions with off-chain (front-end or back-end) code are not examined other than to consider entry points for the contracts, i.e., calls into a smart contract that may disrupt the contract’s functioning.

We should emphasize that, independently of any audit, thoroughly testing all contracts is essential for discovering functional bugs that often also lead to security vulnerabilities. We believe that a large number of the issues listed below could have been discovered by a comprehensive test suite.

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 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”, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

No critical severity issues were found.

HIGH SEVERITY:

ID	Description	STATUS
H1	In LibVault, the _withdraw function can return without transferring the funds to the caller.	**RESOLVED**
In LibVault, the `_withdraw()` function can terminate prematurely under certain conditions without performing the transfer of funds to the caller. This occurs because the contract returns when it contains enough USDC to satisfy the withdrawal request. This is possible if the contract has accumulated fees in USDC		

during previous withdrawal requests, or if the contract has accumulated additional USDC as a result of it fetching an extra 0.5% from the strategies during previous withdrawal requests.

```
function _withdraw(address owner, uint256 oneUSDAmount) internal
returns(uint256) {

    // Ideally the contract will give USDC 1:1 with the 1USD burned
    uint256 usdcToWithdraw =
        LibUtils._convertFromDecimalsToDecimals(oneUSDAmount,
LibStorage.ONE_USD_DECIMALS, LibStorage.USDC_DECIMALS);

    uint256 contractBalance =
LibUtils._balanceOfERC20(LibStorage.USDC_CONTRACT, address(this));

    if(contractBalance >= usdcToWithdraw) {
      return usdcToWithdraw;
    }
```

H2	MasterChefStrategy assumes 1:1 price ratio of stablecoins	**RESOLVED**

`MasterChefStrategy::usdcToUnderlyingAndInvest` adds liquidity to a Uniswap pair at a 1:1 ratio (probably assuming that both tokens are pegged to the USD).

```
function usdcToUnderlyingAndInvest(uint256 amount, bytes32 position) {
  ...

  uint256 halfToken0 = amount / 2;
  uint256 halfToken1 = amount - halfToken0;

  // We swap the USDC for each token needed for the LP Pair.
  uint256 token0Amount = LibUtils._routerSwap(router,
LibStorage.USDC_CONTRACT, token0, halfToken0);
  uint256 token1Amount = LibUtils._routerSwap(router,
```

```
LibStorage.USDC_CONTRACT, token1, halfToken1);

  (, , uint256 liquidity) =
IUniswapV2Router02(mcs.dexRouter).addLiquidity(
      token0,
      token1,
      token0Amount,
      token1Amount,
      1, // We are willing to take whatever the pair gives us
      1, // We are willing to take whatever the pair gives us
      address(this),
      block.timestamp
  );
```

Recent history has shown that stablecoins can get depegged; in such a scenario this action can lead to a loss of funds. It is recommended to add liquidity following the current ratio of the pair (or revert the operation if the current ratio is not close to 1:1).

H3	LibVault::_withdraw function mixes fees and user funds, with no proper accounting	**RESOLVED**

When a user calls the `redeem()` function of the VaultFacet, the call is resolved to the `_withdraw()` function of the LibVault library.

This function will check whether the contract has enough USDC to honour the redemption requests, and if not it will withdraw the shortfall of funds from the strategies - plus 0.5% extra. The function will also keep up to 2% of the withdrawn USDC as a fee, holding them in the contract.

However it seems that no accounting is being used to keep track of what is a fee and what is a deposit. Moreover, fees are indistinguishable from deposits, hence the next withdrawal request will end up using the fees to honour the request, before withdrawing the remaining amount from the strategies.

This lack of accounting has several issues:

- Someone will need to keep track of fees off-chain and only withdraw what is an actual fee, which is an error prone process.
- Since fees are not kept separately from user funds, it is impossible for the owner to withdraw the fees without closing the vault and removing the funds from a strategy. Again this is an error prone process.

We recommend implementing proper accounting for the fees, and storing them separately, so that the owner can easily withdraw them. If it is desired for the fees to remain invested, a solution could be to store them in 1USD form, and transfer them to some fee-holder account, who can later redeem them similarly to all users.

MEDIUM SEVERITY:

No medium severity issues were found.

LOW SEVERITY:

ID	Description	STATUS
L1	Incorrect comparison operator in LibRequirements enforceValidAmountToDeposit function	**RESOLVED**

In LibRequirements, the function `enforceValidAmountToDeposit()` reverts when amountUSDC < `10_000_000`, but the comment above indicates that it should revert if amountUSDC <= `10_000_000`. The presence of an error here seems to be reinforced by the implementation of the `enforceIsMoreThan10USDC()` function, which performs a similar check.

```
/// @dev Valid amount to deposit must be greater than 10 USDC and smaller
```

```
than the allowed by the state variable maxDepositUSDC.

function enforceValidAmountToDeposit(uint256 amountUSDC) internal view {
    if (amountUSDC > vs().maxDepositUSDC || amountUSDC < 10_000_000) {
      revert Vault_Invalid_Deposit();
    }
  }
```

L2	Use of an arbitrary storage slot for strategyStorage	**DISMISSED**

`LibStorage::strategyStorage` uses an arbitrary storage slot passed as parameter:

```
function strategyStorage(bytes32 position) internal pure returns
(Strategy storage ios) {
  assembly {
      ios.slot := position
  }
}
```

The choice of the actual position is done off-chain when the strategy is created (LibManagement::_createMCStrategy), and many protocol functions take an arbitrary position as a parameter. Although the intention is to always use `calculatePositionForStrategy` for computing the position of strategies, this design is unnecessarily risky for two reasons:

1. It remains possible that the position chosen off-chain is wrong and in conflict with other storage data.
2. More importantly, functions that read from an arbitrary position passed as a parameter are very easy to exploit, if that parameter becomes controllable by an adversary. Consider, for instance, the following function:

```
// contracts/MasterChefStrategy.sol
function withdrawRewardsToRecipient(address recipient, bytes32 position)
external {
```

```
  Strategy storage mcs = ios(position);
  // Save some SLOADs
  address rewardToken = mcs.rewardTokenAddress;

  IMasterChef(mcs.masterChefContract).withdraw(mcs.poolId, 0);

  uint256 rewardBalance = LibUtils._balanceOfERC20(rewardToken,
address(this));

  LibUtils._transferERC20(rewardToken, recipient, rewardBalance);
}
```

`ios(position)` can be anywhere in the contract's storage! This function is delegatecalled, so the adversary cannot call it with arbitrary parameters, but if he could, it would be very easy to exploit it (as well as many other functions with `position` as argument).

We recommend `strategyStorage` to hash its argument together with some string, to ensure that (a) no conflicts exist and (b) by controlling the `position` one cannot read arbitrary storage slots.

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	Rewards can only be withdrawn by Diamond contract owner	**DISMISSED**

While depositors are able to withdraw funds they have deposited by using the redeem() function of the VaultFacet, the rewards generated from the investment of those deposits can only be retrieved by the owner of the Diamond contract.

This is achieved by the owner using the claimRewards() function of the VaultFacet, or through the _deactivateStrategy() of the ManagementFacet. In both cases, the owner specifies a single arbitrary address which receives this reward; the reward is not transferred to the depositors by the contract.

This issue was raised with the OneRing team, who stated that the OneRing team will sell the rewards, get USDC that will be used to mint more 1USD, and then distribute the 1USD to the users using an ERC4626 vault.

The team also indicated that as a mitigation to centralisation issues the owner of the Diamond contract would be a multisig contract.

The part of the system involving the distribution of rewards was out of the scope of this audit; therefore this procedure could not be verified during the audit.

N2	Deactivating strategies can leave the contract undercollateralized	**DISMISSED**

Strategies can be deactivated by the owner, which causes all the underlying funds to be withdrawn by the owner. This could leave the contract in an undercollateralized

state, preventing the users from redeeming their 1USD. There are currently no checks in the contract to guarantee sufficient collateralization, this burden is left to the owner.

In response to this issue the OneRing team has indicated that OneRing is constantly adding strategies and reviewing yield sources and needs to be able to modify LP pairs quickly to stay competitive.

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	Some functions in LibRequirements emit non-informative events	**DISMISSED**
In LibRequirements, the functions `enforceValidMinting()` and `enforceValidWithdraw()` emit the Vault_Too_Much_Slippage event, when the corresponding revert commands are not strictly related to slippage.		
A2	Typo in error name in LibRequirements	**RESOLVED**
LibRequirements defines an error called StategyManager_Invalid_Pool_Info. There is a spelling mistake in the error name. The error is used in lines 23 and 152 of the library.		
A3	Compiler bugs	**INFO**

The code is compiled with Solidity 0.8.9 or higher. For deployment, we recommend no floating pragmas, i.e., a specific version, so as to be confident about the baseline guarantees offered by the compiler. Version 0.8.9, in particular, has some known bugs, which we do not believe to affect the correctness of the contracts.

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.