Furucombo Funds Contracts

Smart Contract Security Assessment

Apr. 21, 2022

ABSTRACT

Dedaub was commissioned to perform an audit on the Furucombo Funds contracts at commit hash 0633b98dee5b4f9aeb2606f5dd506ce0b632a6ad. The audited contracts list is the following:

```
FundProxy.sol
ShareToken.sol
FundProxyStorageUtils.sol
MortgageVault.sol
libraries/FundQuota.sol
libraries/LibParam.sol
libraries/LibFee.sol
libraries/DealingAsset.sol
libraries/Whitelist.sol
libraries/StorageArray.sol
libraries/LibUniqueAddressList.sol
libraries/AaveV2DataTypes.sol
libraries/StorageMap.sol
utils/Errors.sol
utils/DelegateCallAction.sol
utils/OwnableAction.sol
utils/DealingAssetAction.sol
utils/DestructibleAction.sol
utils/FundQuotaAction.sol
FundProxyFactory.sol
TaskExecutor.sol
FundImplementation.sol
ComptrollerImplementation.sol
actions/SetupAction.sol
actions/ActionBase.sol
actions/furucombo/AFurucombo.sol
actions/furucombo/IFurucombo.sol
ComptrollerProxy.sol
ComptrollerProxyAdmin.sol
modules/ManagementFeeModule.sol
modules/PerformanceFeeModule.sol
modules/AssetModule.sol
modules/ShareModule.sol
modules/ExecutionModule.sol
FundProxyStorage.sol
assets/AssetResolverBase.sol
```

```
assets/oracles/Chainlink.sol
assets/resolvers/canonical/RCanonical.sol
assets/resolvers/curve/ICurveLiquidityPool.sol
assets/resolvers/curve/RCurveStable.sol
assets/resolvers/aavev2/RAaveProtocolV2Debt.sol
assets/resolvers/aavev2/RAaveProtocolV2Asset.sol
assets/resolvers/uniswapv2like/RUniSwapV2Like.sol
assets/AssetRegistry.sol
assets/AssetRouter.sol
furucombo/Config.sol
furucombo/Storage.sol
furucombo/FurucomboProxy.sol
furucombo/FurucomboRegistry.sol
furucombo/lib/LibCache.sol
furucombo/lib/LibParam.sol
furucombo/lib/LibStack.sol
furucombo/handlers/curve/HCurve.sol
furucombo/handlers/paraswapv5/HParaSwapV5.sol
furucombo/handlers/quickswap/HQuickSwap.sol
furucombo/handlers/quickswap/libraries/UniswapV2Library.sol
furucombo/handlers/aavev2/IFlashLoanReceiver.sol
furucombo/handlers/aavev2/IVariableDebtToken.sol
furucombo/handlers/aavev2/HAaveProtocolV2.sol
furucombo/handlers/aavev2/IStableDebtToken.sol
furucombo/handlers/sushiswap/HSushiSwap.sol
furucombo/handlers/sushiswap/libraries/SushiSwapLibrary.sol
furucombo/handlers/funds/HFunds.sol
furucombo/handlers/HandlerBase.sol
```

Two auditors, as well as two trainees, worked on the codebase over three weeks.

Setting & Caveats

The audited codebase is of rather large size, ~6KLoC,

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 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.

Architecture and High-Level Recommendations

The Furucombo Funds protocol allows for users to become fund managers, i.e. create, parametrize and handle their own fund, as well as investors by purchasing shares of any active fund of interest.

The Furucombo Fund contract is designed in a modular fashion comprised of five modules:
1) *AssetModule*: maintains a list of the fund's assets
2) *ExecutionModule*: entry point for the execution of a strategy
3) *ManagementFeeModule*: handles the management fee which is calculated using an exponential formula on the total supply of the share token
4) *PerformanceFeeModule*: handles the performance fee which is a fraction of the fund's total profits accrued per specific time period. The profits are allowed to be requested by the end of each period (crystallization period), which is defined by the owner upon the construction of the fund.
5) *ShareModule*: handles the investors' deposits and shares

Technically, the most sensitive contracts lie behind an upgradeable proxy (Transparent proxy for the Comptroller and Beacon for the Fund). The protocol utilizes the DSProxy pattern so that a Fund - through its DSProxy Vault which acts like its digital wallet - can trigger the batch execution of a strategy's actions. Because of the way that DSProxy Vaults are built for each Fund, it is possible that the system is DoS attacked so as not to be able to create new Funds, as described in issue H1.
The whole protocol is built on top of the Furucombo engine which is capable of efficiently forwarding static and dynamic information from one action to the next within a strategy's execution flow, because of the Chained Input design.

Due to the high level of automation and dependence on low level operations we strongly advise for thorough testing. At this point, the Furucombo team has an extended testing suite.

From a security standpoint, there are two main protocol entry points that should be considered:

i) the interaction of investors with the Fund.
Investors interact with the fund mainly to purchase and redeem shares. For purchasing shares a deposit of a specific denomination token is required. For successfully redeeming one's shares, the protocol should have enough reserves of the denomination asset. If it doesn't, because most of the value is invested in external protocols, the redeemer either accepts to be put in a "waiting list", and also to suffer a penalty, or the redeem request is canceled. When pending redeem actions exist, investors who purchase shares are given a reward, essentially forwarded by the redeemers' penalty. This leaves space for gaming the system for profit, as we describe in issue M1.

ii) manager interaction with the Fund, in essence, fund creation as well as strategies definition and execution.
The manager of a fund is a semi-trusted entity who creates the fund and sets values to its crucial parameters, such as management/performance fee rates and crystallization period. The protocol takes several security precautions to control manager's privileged access to specific functionality, such as reentrancy guards and rules for fees' calculation and harvesting. On the same note, the fund's parameterization takes place at the creation of the fund and can be altered only during the Review phase. After Finalizing, the fund's most crucial parameters cannot be altered, thus mitigating the corresponding centralization issues. A fund's manager is also responsible for creating investing strategies and triggering their execution within the fund's context. A fund manager profits from management and performance fees, while loses her initially deposited

collateral (mortgage amount) in case of liquidation, however she is not trusted for respecting the investors' deposits. The mortgage amount is supposed to be deposited to a separate contract (MortgageVault) upon the Fund's creation. Due to unguarded handling of the manager's allowance approval towards the MortgageVault, the mortgage amount can be stolen by an attacker as described in C1.

At this point of development, most of the potential sensitive entry points to the system regarding strategy execution allow participation only to whitelisted entities. More specifically, there are currently whitelists for the fund creators, a strategy's actions and assets involved. The protocol is highly abstract and flexible regarding strategy construction, allowing any possible combination of (whitelisted) external protocols. Due to this level of abstraction but also the complexity of the Furucombo engine's specifics, upon which the Fund's Protocol is built, we need to stress out the importance of the whitelists for the system's overall security.

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	Stealing of mortgage amount is possible	**RESOLVED**

`FundProxyFactory::createFund` creates a new fund on behalf of msg.sender. The process requires some amount to be transferred as mortgage from the fund creator to the `MortgageVault` contract. This transfer is triggered within `FundProxyFactory::createFund` by calling the `MortgageVault::mortgage` function right after the `FundProxy` contract has been created:

```
/// FundProxyFactory
function createFund(
    IERC20Metadata denomination_,
    uint256 level_,
    uint256 mFeeRate_,
    uint256 pFeeRate_,
    uint256 crystallizationPeriod_,
    uint256 reserveExecutionRate_,
    string memory shareTokenName_
  ) external returns (address) {
    // ...
    IMortgageVault mortgageVault = comptroller.mortgageVault();

    IFund fund = IFund(address(new FundProxy(address(comptroller),
data)));
    // Dedaub: triggers a transferFrom
    mortgageVault.mortgage(msg.sender, address(fund), amount);
    // ...
}

/// MortgageVault
function mortgage(
    address sender_,
    address fund_,
    uint256 amount_
  ) external {
    Errors._require(fundAmounts[fund_] == 0,
Errors.Code.MORTGAGE_VAULT_FUND_MORTGAGED);
    fundAmounts[fund_] += amount_;
    totalAmount += amount_;
    //Dedaub: anyone can take advantage of the approval required for this transfer
    mortgageToken.safeTransferFrom(sender_, address(this), amount_);
    emit Mortgaged(sender_, fund_, amount_);
}
```

Supposing that the creator of the Fund is an EOA, an allowance approval to `MortgageVault` should take place in a *separate* transaction *before* the fund's creation. An attacker could take advantage of this approval as follows:

- monitor the network for approve transactions towards the MortgageVault
- call MortgageVault::mortgage(sender_, fund_, amount_) with the victim's address as sender_, the approved amount as amount_ and her own arbitrary fund contract address as fund_. Frontrun a createFund transaction if needed.
- call MortgageVault::claim from within her fund contract to receive the stolen amount.

As a solution, msg.sender could be checked against the FundProxyFactory in MortgageVault::mortgage and revert in case of any other caller.

HIGH SEVERITY:

ID	Description	STATUS
H1	DoS on FundProxyFactory::createFund	**RESOLVED**

FundProxyFactory::createFund is used to create new FundProxy instances. When the newly created FundProxy instance is initialized, it creates a DSProxy instance to store the fund's assets.The DSProxy is created via Maker's standard ProxyRegistry::build.

```
function build() public returns (address payable proxy) {
      proxy = build(msg.sender);
}

function build(address owner) public returns (address payable proxy) {
      require(proxies[owner] == DSProxy(0) || proxies[owner].owner() !=
owner); // Not allow new proxy if the user already has one and remains
being the owner
      proxy = factory.build(owner);
      proxies[owner] = DSProxy(proxy);
}
```

There are two interesting facts about the above function:

1. It requires that a DSProxy instance **does not already exist** for the same owner.
2. One can create a DSProxy for any owner, by using the build(owner) variant that takes the owner as an argument.

FundImplementation::initialize expects that no DSProxy exists, which is true under normal operation since the FundProxy instance being initialized is brand new. So it makes no checks, assuming that ProxyRegistry::build will succeed. However, an adversary can cause this condition to fail, as follows:

1. First, he predicts the address of the future FundProxy instance.
2. Then, he creates a DSProxy owned by the predicted address himself, before the FundProxy is even created!
3. When FundImplementation::initialize, is called ProxyRegistry::build will fail since a DSProxy instance already exists, causing the whole FundProxyFactory::createFund transaction to revert.

Predicting the FundProxy's address is actually not hard: contract addresses are generated as hash(sender, nonce). Here the sender is known (FundProxyFactory) and the nonce of a contract increases only on contract creations, so it rarely changes and can be easily predicted. The following code generates a predicted address given the FundProxyFactory and its current nonce.

```
address predicted = address(uint160(uint256(
    keccak256(abi.encodePacked(
       bytes1(0xd6), bytes1(0x94), fundProxyFactory, bytes1(nonce+1)
    ))
)));
```

An adversary could also repeat the process for the next, say, 100 nonces, and create DSProxies for all of them to make sure that the DoS succeeds.

We made a proof of concept implementation of this attack, as a unit test, and verified that it can indeed succeed. The code will be made available together with this report.

As a solution, we recommend querying `ProxyRegistry.proxies` for an existing `DSProxy` instance, and use that if it exists.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	An investor has incentives to force the fund in a pending state	**RESOLVED**

Consider a situation in which an investor (or a group of colluding investors) have shares that exceed the fund's total reserve in denomination token. Then, assume that the fund is in the Executing state and some honest investor calls `redeem`, with `acceptPending_=true` (the investor does not want redeem to fail even if the fund enters the Pending state before the transaction is executed).

Now the malicious investor has financial incentive to sandwich the redeem transaction within two of his own transactions, as follows:

- First, the malicious investor calls `redeem` for an amount equal to the total reserve plus 1 wei. This causes the fund to enter the Pending state, but the malicious investor only pays a penalty for the 1 wei (the remaining shares are covered by the reserve).
- Then the honest investor's `redeem` is executed, causing him to pay a penalty for the whole amount.
- Then the malicious investor `purchases` again his shares, earning the penalty paid by the honest investor.

The problem here is the unfair split of the penalty: both investors redeem (and, in fact, the malicious investor's amount can be much greater than the honest one's), but only the honest one pays a penalty, just because he came second. We don't have a concrete solution to this issue, but we believe that some incentives should be employed to prevent such behavior.

Note, also, that in case the honest user sets `acceptPending_=false`, the same technique can be used to prevent him from redeeming (turning it into a DoS attack). The fund owner might have incentives to perform such a DoS (acting as a malicious investor himself), in order to prevent honest investors from removing their assets from the fund.

LOW SEVERITY:

ID	Description	STATUS
L1	Incentives of funds' liquidations	**ACKNOWLEDGED/ RESOLVED**

Anyone can liquidate a fund that has remained in Pending state for a long time. However, the liquidator does not receive any reward for doing so, while, on the contrary, she will have to spend some money on the gas fees.
In order to incentivise external players to monitor the state of the funds and help keep the system healthy, we suggest that liquidation rewards be introduced, as is the usual case. For example, a portion of the fund manager's mortgage deposit could be given as a reward to the liquidator.

A related (but independent) issue: the fund's mortgage is used as an incentive for the owner to properly close the fund: only then he gets back his mortgage. However, in case of liquidation, although the `pendingLiquidator` becomes the owner, the mortgage is immediately transferred to `comptroller.owner()`. As a consequence, the `pendingLiquidator` has no financial incentives to properly close the fund and allow the investors to recover their funds (the `pendingLiquidator's` incentive is only to preserve the protocol's reputation). We believe that investors would have better trust in the protocol if the mortgage remains locked during liquidation. The `pendingLiquidator` will simply close the fund and receive the mortgage, the same way the original owner would.

L2 Issues with the `_isReserveEnough` check — **RESOLVED**

`FundImplementation::_afterExecute` uses the `_isReserveEnough` check to ensure that the fund has enough reserves after the operation is executed. Although we don't see any major threat, we would like to point out two issues that are worth keeping in mind:

1. From a security standpoint, this check does not really protect against a malicious fund owner. If the owner wants to execute some operation that depletes the fund's reserve, he can workaround this check as follows:
    - He can purchase shares of the fund himself, increasing the reserve
    - He can now execute the operation of interest
    - And finally he redeem the shares

    The effect of this sequence is the same as executing the operation without the `_isReserveEnough` check. Performing the check might still be meaningful to avoid *accidentally* depleting the reserve, but not as a security measure.

2. If the fund enters a Pending state, the owner will need to execute some operations that buy denomination tokens in order to restore the reserve. But since `_isReserveEnough` is called after every `execute`, the owner needs to restore the reserve in a **single** `execute` (any operation that does not restore the reserve will fail).
    Can we be sure that the reserve can be always restored in a single operation, no matter what kind of investments the owner has made? Maybe the protocol's flexible multi-operation execution mechanism guarantees that this is indeed possible. But this assumption is still risky, especially if we consider that performing multiple operations in a single transaction is also subject to gas limits.
    If this is not possible, then the funds would get stuck. A solution, of course, could be to restore the reserve by purchasing more shares. But the owner might not have sufficient funds to do so.
    A solution could be to consider relaxing this check. For instance, when the fund is in pending state, it could only require that the reserve *increases* (even if it is not fully restored).

L3	`assetList` ‘s length should be limited	**RESOLVED**
	Several parts of the code perform `for` loops over the fund’s list of assets. If the assetList becomes too large, the gas needed to perform these loops might exceed the block’s gas limit, essentially making the corresponding operations unusable. This could lead to a DoS of crucial parts of the protocol, likely causing a loss of funds. We are not sure how many assets will be available to the fund owner for investing; if the number is small then this is not an issue. Still, in any case we recommend adding some limit to the total number of assets, a simple solution to ensure that this issue never appears in the future.	

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	`calcAssetValue`: avoid resolver when `asset == quote`	**RESOLVED**
	Since the denomination token always belongs to the asset list, `calcAssetValue` will be often called to convert an amount of denomination tokens to itself (i.e. `asset` would be the same as `quote`). It would be simpler and more gas efficient to just return the amount in this case, without calling the resolver at all.	
A2	Code simplification	**RESOLVED**
	1. In `PerformanceModule::_updatePerformanceFee` the calculation of gross share’s latest price is performed twice, first to be assigned to local variable `grossSharePrice64x64` and second to variable `lastGrossSharePrice64x64`:	

```
function _updatePerformanceFee(uint256 grossAssetValue_) internal virtual
{
    // ...
    int128 grossSharePrice64x64 = grossAssetValue_.divu(totalShare);
    // ...
    // Dedaub: could simply set equal to grossSharePrice64x64 to save gas
    lastGrossSharePrice64x64 = grossAssetValue_.divu(totalShare);
}
```

Since values `grossAssetValue_`, `totalShare` and `grossSharePrice64x64` do not alter their values within the code between those two assignments, `lastGrossSharePrice64x64` can be simply assigned the local variable to simplify the code and save gas.

2. In LibParam::_isStatic and LibParam::_isReferenced the `else` keyword can be omitted:

```
function _isStatic(bytes32 conf_) internal pure returns (bool) {
    if (conf_ & _STATIC_MASK == 0) return true;
    // Dedaub: else keyword is redundant
    else return false;
}

function _isReferenced(bytes32 conf_) internal pure returns (bool) {
    if (_getReturnNum(conf_) == 0) return false;
    // Dedaub: else keyword is redundant
    else return true;
}
```

A3	Dead code	RESOLVED

In FundImplementation::close there is a redundant if statement which can be removed:

```
function close() public override onlyOwner nonReentrant
```

```
  whenStates(State.Executing, State.Liquidating) {
    // Dedaub: if statement is unnecessary as it's repeated in
_settlePendingShare
    if (_getResolvePendingShare(false) > 0) {
        _settlePendingShare(false);
    }
    // ...
}

function _settlePendingShare(bool applyPenalty_) internal {
    uint256 redeemShare = _getResolvePendingShare(applyPenalty_);
    if (redeemShare > 0) {
        // ...
    }
}
```

A4	SetupAction: no need to create a new one for every Fund	RESOLVED

Upon finalizing a Fund's creation, the Vault is triggered by the Fund to set an infinite approval on the Fund for the denomination asset, so as to be able to transfer amounts from the Vault to an investor upon redeeming. A SetupAction contract is delegatecall()ed by the DSProxy Vault for this purpose, which is unique for each Fund and created upon the Fund's construction:

```
constructor(IDSProxyRegistry dsProxyRegistry_) {
        dsProxyRegistry = dsProxyRegistry_;
        setupAction = new SetupAction();
}
```

However, the SetupAction contract carries no fund-specific information. It only defines the `maxApprove` function:

```
contract SetupAction is ISetupAction {
    using SafeERC20 for IERC20;
    function maxApprove(IERC20 token_) external {
        token_.safeApprove(msg.sender, type(uint256).max);
    }
}
```

There seems to be no reason to create a new setup contract for each Fund. Instead, and in respect to the system's architecture, the setup contract could be only once deployed and its address stored in the Comptroller contract to reference the setupAction, just like TaskExecutor is stored as execAction. With this design the code would be simplified and save on gas costs.

A5	Floating pragma	ACKNOWLEDGED

The floating pragma ˆ0.8.0 is used in some contracts, allowing them to be compiled with versions 0.8.0 - 0.8.12 of the Solidity compiler. Although the differences between these versions are small, floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contracts' deployment.

A6	Compiler known issues	INFO

Solidity compiler versions v0.8.0, v0.8.10 have, at the time of writing, some known bugs. We inspected the code and found that it is not affected by these bugs.

DISCLAIMER

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.