Adapters V3

Overview

**Adapters**

**Overview**

Adapters are smart contracts that allow Gearbox users to interact with other protocols. For every credit manager and every *target contract* we want accounts from this manager to be able to interact with (e.g., Uniswap router, Yearn vault, Convex booster, etc.), there must be an *adapter contract* registered in the system.

Adapters are wrappers around target contracts, with similar interfaces, whose main task is to execute the call from the credit account to the target. However, there are other things they perform under the hood:

* handle token approvals;
* enable or disable tokens as credit account's collateral;
* validate or change call parameters if it is needed to ensure funds safety;
* in some cases, they might even tokenize the result of the operation as Gearbox only recognizes ERC-20 tokens as collateral.

**New security paradigm**

Third version of adapters comes with a series of security enhancements compared to its predecessor:

* adapters can now only be called as part of the multicall, which runs the full collateral check after all operations (previously, adapters were allowed to be called directly, which would trigger an inherently less secure fast collateral check);
* adapters now always revoke target contracts' allowances for the credit accounts' tokens after operations;
* adapters aim to minimize arbitrary code execution by disallowing interaction with arbitrary tokens or contracts.

Note that the new adapters are compatible with old credit managers and pools, so, although they come under the name V3, they can already be used starting from the V2.1 release.

**Using adapters**

Let's consider a simple example: swapping USDC to WETH using Gearbox Uniswap V3 adapter.

First of all, install the required packages: @gearbox-protocol/integrations-v3 and @gearbox-protocol/core-v3 (can be installed via forge or npm).

From these packages, import the relevant interfaces and libraries:

import {ICreditManager} from "@gearbox-protocol/core-v3/contracts/interfaces/ICreditManager.sol";

import {ICreditFacade} from "@gearbox-protocol/core-v3/contracts/interfaces/ICreditFacade.sol";

import {IUniswapV3Adapter} from "@gearbox-protocol/integrations-v3/contracts/interfaces/IUniswapV3Adapter.sol";

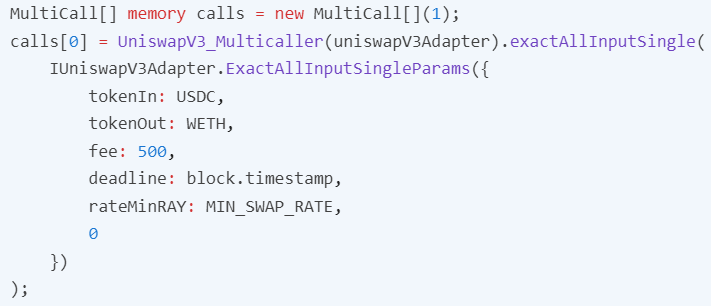
import {UniswapV3\_Multicaller, UniswapV3\_Calls} from "@gearbox-protocol/integrations-v3/contracts/multicall/UniswapV3.sol";

using UniswapV3\_Calls for UniswapV3\_Multicaller;

The address of Uniswap V3 adapter for the given credit manager can be found like this:

address uniswapV3Adapter = ICreditManager(CREDIT\_MANAGER).contractToAdapter(UNISWAP\_V3\_ROUTER);

Now, we need to prepare calldata for the multicall:



This call would try to swap all CA's balance of USDC to WETH through 0.05% pool, and revert if exchange rate is worse than MIN\_SWAP\_RATE. It would also enable WETH and disable USDC as CA's collateral tokens.

Finally, execute the multicall:

ICreditFacade(ICreditManager(CREDIT\_MANAGER).creditFacade()).multicall(calls);

**NOTE**: for V2.1, install @gearbox-protocol/integrations-v3@v2.1 and @gearbox-protocol/core-v2, and replace core-v3 imports with core-v2.

Adapters V3

Building adapters

**Building adapters**

All adapters must pass internal and external security audits, and can only be added to the system with approval from the DAO. Nevertheless, we welcome all the contributors willing to help in integrating Gearbox with as many DeFi protocols as possible. This page lists the guidelines we hope to be useful for them to start writing secure and efficient adapters. It might also help to gain advanced understanding of the system and our reasoning about its security.

**General guidelines**

First of all, since adapters are wrappers, it makes sense to thoroughly understand the protocol you're making a wrapper for. What are the main contracts to interact with? Are they fully compatible with Gearbox?

Keep in mind the following things:

* credit accounts don't work with native ETH, so gateway is needed to unwrap WETH before calling the target contract (e.g., LidoV1Gateway and CEtherGateway);
* pools don't support rebasing tokens, so non-rebasing wrapper is needed if there should be a pool for this token (e.g., WstETH and WrappedAToken);
* Gearbox only recognizes ERC-20 tokens as collateral, so if the protocol itself doesn't tokenize the operation, the adapter should do that instead (e.g., ConvexStakedPositionToken used in Convex BaseRewardPool adapter);
* if adapter produces a token that has no Chainlink price feed (LP tokens, non-rebasing wrappers, etc), a custom price feed must be created for this token (this is explained in more detail in the [oracles](https://dev.gearbox.fi/oracle/overview) section).

Now, what functionality should be in the adapter?

* wrappers for all target contract functions that can modify account's state (interface must be the same, but returned values can be omitted);
* versions of those functions that operate on the entire account's balance, unless they already exist in the target contract;
* wrappers for state-reading functions should only be added when necessary.

We now need to make wrapping functions secure. In order to do that, beyond simply calling the target contract, adapter functions must ensure that:

* they are called as part of the multicall and operate on the account on which it is executed;
* target contract approvals for credit account's tokens are revoked after the operation;
* tokens spent and received during the operation are recognized as collateral by the credit manager;
* ability to execute arbitrary code during the target contract call is minimized;
* tokens recipient is always the credit account.

Finally, for every adapter, there should be a library that would prepare calldata for multicalls.

**AbstractAdapter**

AbstractAdapter is a helper contract that provides utility functions to securely interact with credit account, credit manager and target contract. It should be inherited and used by all adapters.

Let's analyze the functionality it provides:

* addressProvider: Contract that allows to access global addresses like Gearbox treasury, WETH, etc.
* configuratorOnly: modifier that ensures that function is called by the configurator. This modifier should be used for all functions that can change adapter's configuration parameters.
* \_creditFacade: returns the credit facade connected to the adapter's credit manager.
* creditFacadeOnly: modifier that ensures that function is called by the credit facade, which is only possible during the multicall. This modifier should be used for all functions that modify account's state. Although adapters would revert if called not from the multicall because credit account is not owned by the facade, it serves as re-entrancy protection in case attacker somehow gains control during the target contract call.
* \_creditAccount: returns the credit account the multicall is executed on, which is the account currently owned by the credit facade. This function should always be used when adapter needs to know the address of the credit account it's called for.
* \_getMaskOrRevert: checks that token is registered as collateral token in the credit manager and returns its mask. This function can be used to check and initialize token masks in adapter's constructor and later use them in \_changeEnabledTokens.
* \_approveToken: checks that token is registered as collateral token in the credit manager and approves given amount of credit account's tokens to the target contract.
* \_enableToken: checks that token is registered as collateral token in the credit manager and enables it as collateral of the credit account.
* \_disableToken: disables token as collateral of the credit account.
* \_changeEnabledTokens: enables and disables multiple tokens by their token masks in the credit manager in a single call.
* \_execute: calls the target contract from the credit account with passed calldata and returns the bytes-encoded call result.
* \_executeSwapNoApprove: same as \_execute, but also checks and (optionally) disables the input token, and checks and enables the output token. It is useful for swap operations when input and output tokens are not known in advance.
* \_executeSwapSafeApprove: same as \_executeSwapNoApprove, but also gives the target contract infinite approval for the input token before the call and resets it to 1 after the call. It is useful for approve-requiring swap operations when input and output tokens are not known in advance.

**Optimizations**

Here are some additional optimizations that can be made in wrapping functions:

* if wrapping function doesn't process or modify parameters in any way, pass msg.data directly to \_execute... (saves gas);
* when spending the entire balance, spend balance - 1 instead of balance (saves gas);
* when revoking an approval, set the allowance to 1 instead of 0 (saves gas);
* enable tokens received after the operation, disable tokens whose entire balance is spent after the operation (simplifies multicalls for users);
* when tokens that are spent/received in the operation can be known in advance, use \_changeEnabledTokens to perform all enabling/disabling in a single call (saves gas).

**Checklist**

Keeping all written above in mind, we can create a formal set of conditions that adapters must satisfy:

*  Adapter must be made compatible with Gearbox protocol;
*  Adapter must inherit and make use of AbstractAdapter;
*  All wrapping functions that modify account's state must have the creditFacadeOnly modifier;
*  All wrapping functions can only modify the state of the \_creditAccount();
*  All wrapping functions that allow to specify a recipient must set it to the \_creditAccount();
*  All wrapping functions that require token approval to execute an operation must reset it to 1 after;
*  All wrapping functions that receive/spend tokens must call \_enableToken()/\_disableToken() on them (or \_changeEnabledTokens if masks were initialized in the constructor).

On the next page, we'll try to write a generic adapter for ERC-4626 vaults and evaluate it against this checklist.

Adapters V3

Tutorial: writing ERC-4626 adapter

# Tutorial: writing ERC-4626 adapter

In this tutorial, we'll write an adapter for ERC-4626 vaults.

First of all, let's understand our target contract. We'll use the OpenZeppelin's [implementation(opens in a new tab)](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC4626.sol) as a reference.

Do we need to make any adjustments to make the contract compatible with Gearbox protocol?

1. We'll assume that vault's underlying asset is a typical non-rebasing ERC-20 token, already supported by the credit manager, so no gateways are needed.
2. Vault contract itself is an ERC-20 token representing user's share in the vault, so we only need to get this token added to the credit manager by the DAO.
3. Likely, there's no Chainlink oracle for the vault share, however LPPriceFeed contract can help us, which computes share's price as product between Chainlink price of underlying and exchange rate between share and underlying (the latter can be found using convertToAssets function).

## Interface

Let's now figure out what interface the adapter should have.

1. There are four main state-modifying functions in the contract: deposit, mint, withdraw, and redeem, and our adapter must also implement them. We can also add depositAll and redeemAll versions that would deposit all balance of underlying asset and redeem all balance of shares, respectively.
2. Most state-reading functions should be called directly from the target contract. It only makes sense to add the asset() function which returns underlying asset address.
3. Note that we know both underlying asset and vault's share addresses at the moment of adapter creation, so we can store their collateral masks and perform gas-optimized enabling/disabling. Let's make those masks public and add shareMask() and assetMask() functions to the interface.

Eventually, the adapter interface might look as follows:

interface IERC4626Adapter is IAdapter { /// @notice Vault's underlying asset function asset() external view returns (address); /// @notice Collateral token mask of vault's share in the credit manager function shareMask() external view returns (uint256); /// @notice Collateral token mask of vault's underlying asset in the credit maanger function assetMask() external view returns (uint256); /// @notice Deposits given amount of underlying asset into the vault to mint vault's shares /// @param assets Amount of underlying asset to deposit function deposit(uint256 assets, address) external; /// @notice Deposits the entire balance of underlying asset into the vault to mint vault's shares, /// disables underlying asset function depositAll() external; /// @notice Deposits underlying asset into the vault to mint given amount of vault's shares /// @param shares Amount of shares to mint function mint(uint256 shares, address) external; /// @notice Redeems vault's shares to withdraw given amount of underlying asset /// @param assets Amount of underlying asset to withdraw function withdraw(uint256 assets, address, address) external; /// @notice Redeems given amount of vault's shares to withdraw underlying asset /// @param Amount of shares to redeem function redeem(uint256 shares, address, address) external; /// @notice Redeems the entire balance of vault's shares to withdraw underlying asset, /// disables vault's shares function redeemAll() external;}

It's worth noting that return values from original interface are omitted, and receiver/owner parameters are ignored because they must always be set to the credit account address.

## Implementation

With understanding what the adapter should do, it's time to implement it.

### Constructor

Let's start with contract constructor and state variables.

1. All adapters must inherit AbstractAdapter to gain access to credit manager functionality. Our adapter should also implement the interface we prepared above.
2. Adapter type should be added to AdapterType enum in core repository. You can use AdapterType.ABSTRACT until then.
3. We can make asset, shareMask and assetMask public immutable variables as they can't change during adapter's lifetime, and storing them in such a way is more gas-efficient.
4. All adapter constructors should take at least two parameters: credit manager address and target contract address. We then use them to initialize abstract adapter. Finally, we initialize state variables.

contract ERC4626Adapter is AbstractAdapter, IERC4626Adapter { AdapterType public constant override \_gearboxAdapterType = AdapterType.ERC4626; uint16 public constant override \_gearboxAdapterVersion = 1; /// @inheritdoc IERC4626Adapter address public immutable override asset; /// @inheritdoc IERC4626Adapter uint256 public immutable override shareMask; /// @inheritdoc IERC4626Adapter uint256 public immutable override assetMask; /// @notice Constructor /// @param \_creditManager Address of the credit manager to connect the adapter to /// @param \_vault Address of the target vault contract constructor(address \_creditManager, address \_vault) AbstractAdapter(\_creditManager, \_vault) { asset = IERC4626(\_vault).asset(); shareMask = \_getMaskOrRevert(\_vault); assetMask = \_getMaskOrRevert(asset); }}

### Wrapping functions

Now let's actually implement the state-modifying functionality.

Let's consider a step-by-step process of writing a wrapper function, keeping in mind abstract adapter's helper functions.

**Step 1**. Call the wrapped function of the target contract with passed calldata.

function deposit(uint256 assets, address receiver) external override { \_execute(abi.encodeCall(IERC4626.deposit, (assets, receiver)));}

That would revert because target contract needs an approval to execute the call.

**Step 2.** Handle token approvals: give approval before the operation and revoke after it

function deposit(uint256 assets, address receiver) external override { \_approveToken(asset, type(uint256).max); \_execute(abi.encodeCall(IERC4626.deposit, (assets, receiver))); \_approveToken(asset, 1);}

We could've just called \_approveToken(asset, assets + 1) and probably gotten the same result, but let's remember how adversarial our environment is: we recommend setting the approval to 1 explicitly.

Okay, this would do the job, but is it safe? Nope, it transfers shares to an arbitrary address!

**Step 3.** Ensure that tokens recipient is always the credit account.

function deposit(uint256 assets, address) external override { address creditAccount = \_creditAccount(); \_approveToken(asset, type(uint256).max); \_execute(abi.encodeCall(IERC4626.deposit, (assets, creditAccount))); \_approveToken(asset, 1);}

Are we done? Hell no! A potential attacker would be able to call the function once again if he manages to execute his own code during target contract call!

**Step 4.** Always add the creditFacadeOnly modifier.

function deposit(uint256 assets, address) external override creditFacadeOnly { address creditAccount = \_creditAccount(); \_approveToken(asset, type(uint256).max); \_execute(abi.encodeCall(IERC4626.deposit, (assets, creditAccount))); \_approveToken(asset, 1);}

This function is good now, but we should still take care of the user and simplify multicalls for them.

**Step 5.** Enable tokens received during the call and disable the ones that were fully spent.

function deposit(uint256 assets, address) external override creditFacadeOnly { address creditAccount = \_creditAccount(); \_approveToken(asset, type(uint256).max); \_execute(abi.encodeCall(IERC4626.deposit, (assets, creditAccount))); \_approveToken(asset, 1); \_changeEnabledTokens(shareMask, 0);}

Asset is not disabled because this function doesn't generally spend the entire balance.

Notice that this step is as important for security as the previous ones in case when function deals with arbitrary tokens, because it checks whether tokens are recognized by the system.

### Full contract

Let's repeat the same process for other wrapping functions. After some refactoring and gas optimization, the contract would look like this:

contract ERC4626Adapter is AbstractAdapter, IERC4626Adapter { AdapterType public constant override \_gearboxAdapterType = AdapterType.ERC4626; uint16 public constant override \_gearboxAdapterVersion = 1; /// @inheritdoc IERC4626Adapter address public immutable override asset; /// @inheritdoc IERC4626Adapter uint256 public immutable override shareMask; /// @inheritdoc IERC4626Adapter uint256 public immutable override assetMask; /// @notice Constructor /// @param \_creditManager Address of the credit manager to connect the adapter to /// @param \_vault Address of the target vault contract constructor(address \_creditManager, address \_vault) AbstractAdapter(\_creditManager, \_vault) { asset = IERC4626(\_vault).asset(); shareMask = \_checkToken(\_vault); assetMask = \_checkToken(asset); } /// @inheritdoc IERC4626Adapter function deposit(uint256 assets, address) external override creditFacadeOnly { \_deposit(\_creditAccount(), assets, false); } /// @inheritdoc IERC4626Adapter function depositAll() external override creditFacadeOnly { address creditAccount = \_creditAccount(); uint256 balance = IERC20(asset).balanceOf(creditAccount); if (balance <= 1) return; unchecked { \_deposit(creditAccount, balance - 1, true); } } /// @dev Implementation of deposit function \_deposit(address creditAccount, uint256 assets, bool disableAsset) internal { \_approveToken(asset, type(uint256).max); \_execute(abi.encodeCall(IERC4626.deposit, (assets, creditAccount))); \_approveToken(asset, 1); \_changeEnabledTokens(shareMask, disableAsset ? assetMask : 0); } /// @inheritdoc IERC4626Adapter function mint(uint256 shares, address) external override creditFacadeOnly { address creditAccount = \_creditAccount(); \_approveToken(asset, type(uint256).max); \_execute(abi.encodeCall(IERC4626.mint, (shares, creditAccount))); \_approveToken(asset, 1); \_changeEnabledTokens(shareMask, 0); } /// @inheritdoc IERC4626Adapter function withdraw(uint256 assets, address, address) external override creditFacadeOnly { address creditAccount = \_creditAccount(); // NOTE: no approval needed since target contract will be called by CA \_execute(abi.encodeCall(IERC4626.withdraw, (shares, creditAccount, creditAccount))); \_changeEnabledTokens(assetMask, 0); } /// @inheritdoc IERC4626Adapter function redeem(uint256 shares, address, address) external override creditFacadeOnly { \_redeem(\_creditAccount(), shares, false); } /// @inheritdoc IERC4626Adapter function redeemAll() external override creditFacadeOnly { address creditAccount = \_creditAccount(); uint256 balance = IERC20(targetContract).balanceOf(creditAccount); if (balance <= 1) return; unchecked { \_redeem(creditAccount, balance - 1, true); } } /// @dev Implementation of redeem function \_redeem(address creditAccount, uint256 shares, bool disableShare) internal { // NOTE: no approval needed since target contract will be called by CA \_execute(abi.encodeCall(IERC4626.redeem, (shares, creditAccount, creditAccount))); \_changeEnabledTokens(assetMask, disableShare ? shareMask : 0); }}

## Checklist

As promised, let's evaluate the new adapter against the checklist on the previous page.

*  Adapter must be made compatible with Gearbox protocol — the only adaptation is custom oracle
*  Adapter must inherit and make use of AbstractAdapter — inherits explicitly
*  All wrapping functions that modify account's state must have the creditFacadeOnly modifier — recall step 4
*  All wrapping functions can only modify the state of the \_creditAccount() — recall step 3
*  All wrapping functions that allow to specify a recipient must set it to the \_creditAccount() — recall step 3
*  All wrapping functions that require token approval to execute an operation must reset it to 1 after — recall step 2
*  All wrapping functions that receive/spend tokens must call \_enableToken()/\_disableToken() on them (or \_changeEnabledTokens if masks were initialized in the constructor) — recall step 5 and constructor