Feb 17, 2023 8 min read

ERC4626 Interface Explained

Updated: Jan 29

ERC4626 is a tokenized vault standard that uses ERC20 tokens to represent shares of some other asset.

How it works is you deposit one ERC20 token (token A) into the ERC4626 contract, and get another ERC20 token back, call it token S.

In this example, token S represents your share of all of the token A owned by the contract (not the total supply of A, only the balance of A in the ERC4626 contract).

At a later date, you can put token S back into the vault contract and get token A returned to you.

If the balance of token A in the vault grew faster than than token S was produced, you would withdraw proportionately larger amount of token A than what you deposited.

An ERC4626 contract is also an ERC20 token

When an ERC4626 contract gives you an ERC20 token for the initial deposit, it gives you token S (an ERC20 compliant token). The ERC20 token isn't a separate contract. It's implemented in the ERC4626 contract. In fact, you can see this is how OpenZeppelin defines the contract in Solidity:

```
48      * _Available since v4.7._
49      */
50     abstract contract ERC4626 is ERC20, IERC4626 {
51         using Math for uint256;
52
53         IERC20 private immutable _asset;
54         uint8 private immutable _underlyingDecimals;
55
56         /**
57          * @dev Set the underlying asset contract. This must be an ERC20-compatible contract (ERC20 or ERC777).
58          */
59         constructor(IERC20 asset_) {
60             (bool success, uint8 assetDecimals) = _tryGetAssetDecimals(asset_);
61             _underlyingDecimals = success ? assetDecimals : 18;
62             _asset = asset_;
63         }
```

The ERC4626 extends the ERC20 contract and during construction phase, it takes as an argument the other ERC20 token users will be depositing to it.

Therefore, ERC4626 supports all the functions and events you expect from ERC20:

- **balanceOf**
- **transfer**
- **transferFrom**
- **approve**
- **allowance**

And so forth.

This token is referred to as the shares in an ERC4626. This is the ERC4626 contract itself.

The more shares you own, the more rights you have to the underlying asset (the other ERC20 token) that gets deposited into it.

ERC4626 Motivation

Let's use a real example to motivate the design.

Let's say we all own a company, or a liquidity pool, that earns a stablecoin DAI periodically. The stablecoin DAI is the asset in this case.

One inefficient way we could distribute the earnings is to push out DAI to each of the holders of the company on a pro-rata basis. But this would be extremely expensive gas wise.

Similarly, if we were to update everyone's balance inside a smart contract, that would be expensive too.

Instead, this is how the workflow would work with ERC4626.

Let's say you and nine friends get together and each deposit 10 DAI each into the ERC4626 vault (100 DAI total). You get back one share.

So far so good. Now your company earns 10 more DAI, so the total DAI inside the vault is now 110 DAI.

When you trade your share back for your part of the DAI, you don't get 10 DAI back, but 11.

Now there is 99 DAI in the vault, but 9 people to share it among. If they were to each withdraw, they would get 11 DAI each.

Note how efficient this is. When someone makes a trade, instead of updating everyones shares one-by-one, only the total supply of shares and the amount of assets in the contract changes.

ERC4626 does not have to be used in this manner. You can have an arbitrary mathematical formula that determines the relationship between shares and assets. For example, you could say every time someone withdraws the asset, they also have to pay some sort of a tax that depends on the block timestamp or something like that.

The ERC 4626 standard provides a gas efficient means for executing very common DeFi accounting practices.

ERC4626 Shares

Naturally, users want to know which asset the ERC4626 uses and how many are owned by the contract, so there are two solidity functions in the ERC4626 specification for that.

```
function asset() returns (address)
```

The asset function returns the address of the underlying token used for the Vault. If the underlying asset was say, DAI, then the function would return the ERC20 contract address of DAI 0x6b175474e89094c44da98b954eedeac495271d0f.

```
function totalAssets() returns (uint256)
```

Calling the totalAssets function will return the total amount of assets "managed" (owned) by the vault, i.e. the number of ERC20 tokens owned by the ERC4626 contract. The implementation is quite simple in OpenZeppelin:

```
/** @dev See {IERC4626-totalAssets}. */
function totalAssets() public view virtual override returns (uint256) {
    return _asset.balanceOf(address(this));
}
```

There is of course no function to get the shares address, because that is just the address of the ERC4626 contract.

Giving assets, getting shares: deposit() and mint()

Let's copy and paste the two specifications for making this trade directly from the EIP.

```
function deposit(uint256 assets, address receiver) public virtual override returns (uint256)


// EIP: Mints exact number of vault shares to receiver, as specified by user, by calculating
number of required shares of underlying asset.

function mint(uint256 shares, address receiver) public virtual override returns (uint256)
```

According to the EIP, the user is depositing assets and getting shares back, so what's the difference between these two functions?

- **With *deposit()*, you specify how many assets you want to put in, and the function will calculate how many shares to send to you.**
- **With *mint()*, you specify how many shares you want, and the function will calculate how much of the ERC20 asset to transfer from you.**

Of course, if you don't have enough assets to transfer in to the contract, the transaction will revert.

The uint256 that gets returned to you is amount of shares you get back.

The following invariant should always be true

```
// remember, erc4626 is also an erc20 token
uint256 sharesBalanceBefore = erc4626.balanceOf(address(this));
uint256 sharesReceived = erc4626.deposit(numAssets, address(this));

// strict equality checks in accounting are a big no no!
assert(erc4626.balanceOf(address(this)) >= sharesBalanceBefore + sharesReceived);
```

Anticipating how many shares you will get

If you are using web3.js, you can issue a staticcall to deposit or mint functions to predict what will happen. If you are doing this on-chain however, you have the following two functions at your disposal:

- **previewDeposit**
- **previewMint**

Like their state changing counterparts, previewDeposit takes assets as an argument and previewMint takes shares as an argument.

Anticipating how many shares you will get under ideal conditions

Confusingly enough, there is also a view function called convertToShares which takes assets as an argument and returns the amount of shares you will get back under ideal conditions (no slippage or fees).

Why would you care about this ideal information that doesn't reflect the trade you will execute?

The difference between ideal and actual results tells you how much your trade is impacting the market and how the fee depends on trade size. A smart contract could do a binary search on the difference between convertToShares and previewMint to find the best trade size to execute.

Returning shares, getting assets back

The inverse of deposit and mint is withdraw and redeem respectively.
With deposit, you specify the assets you want to trade and the contract calculates how many shares you get.

With mint, you specify how many shares you want, and the contract calculates how many assets to take from you.

Similarly, withdraw lets you specify how many assets you want to take from the contract, and the contract calculates how many of your shares to burn.

With redeem, you specify how many shares you want to burn, and the contract calculates the amount of assets to give back.

The view methods for withdraw and redeem are previewRedeem and previewWithdraw respectively.

The idealized analog of these functions is convertToAssets which takes shares as an argument and gives you how many assets you will get back, not including fees and slippage.

Summary of functions so far

What about the address argument?

```
function mint(uint256 shares, address receiver) external returns (uint256 assets);

function deposit(uint256 assets, address receiver) external returns (uint256 shares);

function redeem(uint256 shares, address receiver, address owner) external returns (uint256
assets);

function withdraw(uint256 assets, address receiver, address owner) external returns (uint256
shares);
```

The functions mint, deposit, redeem, and withdraw, have an second argument “receiver” for cases where the account receiving shares or assets from the ERC4626 is not msg.sender. This means I can deposit assets into the account and specify that the ERC4626 contract give you the shares.

maxDeposit, maxMint, maxWithdraw, maxRedeem

These functions take identical arguments to their state-changing counterparts and return the largest trade they can execute. This can change per address (remember, we just discussed that these functions take addresses as arguments).

Events

ERC4626 has only two events in addition to the ERC20 events it inherits: Deposit and Withdraw. These are also emitted if mint and redeem were called, because functionally the same thing happened: tokens were swapped.

Problems with slippage

Any token swapping protocol has an issue where the user might not get back the amount of tokens they were expecting.

For example, with automated market makers, a large trade might use up the liquidity and cause the price to move substantially.

Another issue is a transaction getting frontrun or experiencing a sandwich attack. In the examples above, we've assumed the ERC4626 contract maintains a one-to-one relationship between asset and shares regardless of the supply, but the ERC4626 standard does not dictate how the pricing algorithm should work.

For example, suppose we make the amount of shares issued a function of the square root of the assets deposited. In that case, whoever deposits first will get a larger amount of shares. This could encourage opportunistic traders to frontrun deposit orders and force the next buyer to pay a larger amount of the asset for the same amount of shares.

The defense against this is simple: the contract interacting with an ERC4626 should measure the amount of shares it received during a deposit (and assets during a withdraw) and revert if it does not receive the quantity expected within a certain slippage tolerance.

This is a standard design pattern to deal with slippage issues. It will also defend against the issue described below.

ERC4626 inflation attack

Although ERC4626 is agnostic to the algorithm that translates prices to shares, most implementations use a linear relationship. If there are 10,000 assets, and 100 shares, then 100 assets should result in 1 share.

But what happens if someone sends 99 assets? It will round down to zero and they get zero shares.

Of course no-one would intentionally throw away their money like this. However, an attacker can frontrun a trade by donating assets to the vault.

If an attacker donates money to the vault, one share is suddenly worth more than it was initially. If there are 10,000 assets in the vault corresponding to 100 shares, and the attacker donates 20,000 assets, then one share is suddenly worth 300 assets instead of 100 assets. When the victim's trade trades in assets to get back shares, they suddenly get a lot fewer shares — possibly zero.

There are three defenses:

- Revert if the amount received is not within a slippage tolerance (described earlier)
- The deployer should deposit enough assets into the pool such that doing this inflation attack would be too expensive
- Add "virtual liquidity" to the vault so the pricing behaves as if the pool had been deployed with enough assets.

Here is OpenZeppelin's implementation of virtual liquidity:

When calculating the amount of shares a depositor receives, the total supply is artificially inflated (at a rate the programmer specifies in _decimalsOffset()).

- **totalAssets() = the balance of assets held by the ERC4626**
- **assets = the amount of assets the user is depositing**

The formula is

```
shares_received = assets_deposited * totalSupply() / totalAssets();
```

There is some implementation details for rounding in favor of the pool and adding 1 to totalAssets() to ensure we don't divide by zero if the pool is empty.

Let's say we have the following numbers:
assets_deposited = 1,000
totalSupply() = 1,000
totalAssets() = 999,999 (the formula adds 1, so we will set it this way to make the number nice)

In that case, the shares the user will get is 1,000 x 1,000 ÷ 1,000,000, or exactly 1.

This is obviously very fragile. If the attacker frontruns the deposit of 1,000 shares and deposits assets, then the victim will get zero back, because 1 million divided by a number larger than 1 million is zero in integer division.

How does virtual liquidity solve this? Using the code from the screenshot above, we would set _decimalOffset() to be 3, so that way totalSupply() gets 1,000 added to it.

Effectively, we are making the numerator 1,000 times larger. This forces the attacker to make a donation 1,000 times as large, which disincentivizes them from conducting the attack.

Real life examples of share / asset accounting

Earlier versions of Compound minted what they called c-tokens to users who supplied liquidity. For example, if you deposited USDC, you would get a separate cUSDC (Compound USDC) back. When you decided to stop lending, you would send back your cUSDC to compound (where it would be burned) then get your pro-rata share of the USDC lending pool.

Uniswap used LP tokens as "shares" to represent how much liquidity someone had put into a pool, (and how much they could withdraw pro-rata) when they redeemed the LP tokens for the underlying asset.

Learn More

Learn more advanced topics in our blockchain bootcamp.

Further Resources

Original EIP Author on Youtube
Openzeppelin's implementation
Solmate implementation

Recent Posts See All

Web3 Blockchain Bootcamp