Jeffrey Scholz Jan 8 7 min read

Understanding Collateral, Liquidations, and Reserves in Compound V3

In this chapter we will examine the following topics about Compound V3:

- collateral valuation
- absorbing insufficiently collateralized loans (liquidations)
- selling absorbed collateral
- what reserves are
- and how reserves affect liquidations.

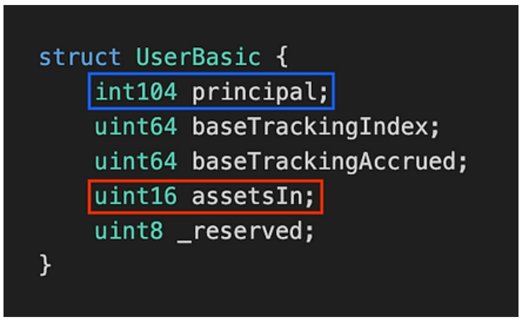
These are a lot of topics to go over in one article, but they are all heavily intertwined, so it's best we cover them in one treatment.

Prerequisites

The reader should already be familiar with how Compound V3 defines principal and present value and with DeFi liquidations and collateral.

UserBasic Storage Struct

Let's revisit the UserBasic struct in CometStorage.sol.



```
struct UserBasic {
    int104 principal;
    uint64 baseTrackingIndex;
    uint64 baseTrackingAccrued;
    uint16 assetsIn;
    uint8 _reserved;
}
```

If the principal (blue box) is negative, it means the user is a borrower, and the negative value will be the ***principal*** value of their debt.

assetsIn (red box) is a bitmap to indicate whether they've deposited an certain collateral asset or not. At this time of writing, the bitmap is laid out as follows:

The variables baseTrackingIndex and baseTrackingAccrued are for bookkeeping the distribution of rewards, and will be discussed in a separate article. The variable _reserved is unused.

Note that this struct does not tell us how much collateral the user is holding. That is held in balance variable the UserCollateral struct which is stored in the userCollateral nested mapping. The _reserved variable is unused.

To list out the collateral a user has supplied, we loop through 0…numAssets Compound stores and check if that bit is set to one for that user. If it is, we obtain the token address associated with that bit and check the user balance in userCollateral[user][collateralAsset] to see how much of that collateral the user holds.

By multiplying the balance with the oracle price, we know the dollar value of the user's collateral. The following table gives an example of summing up the total value of a user's collateral.

AssetInfo

The address of the oracle where Compound obtains the collateral price from is stored in the AssetInfo struct (blue box).

Observe the AssetInfo struct above is 432 bits large — it takes 2 slots to store it. We will revisit this in a follow section.

Displaying AssetInfo on the Compound Finance Markets

Let's compare the content of the AssetInfo struct shown above with the Compound Finance Market UI. Here we see most of the information displayed.

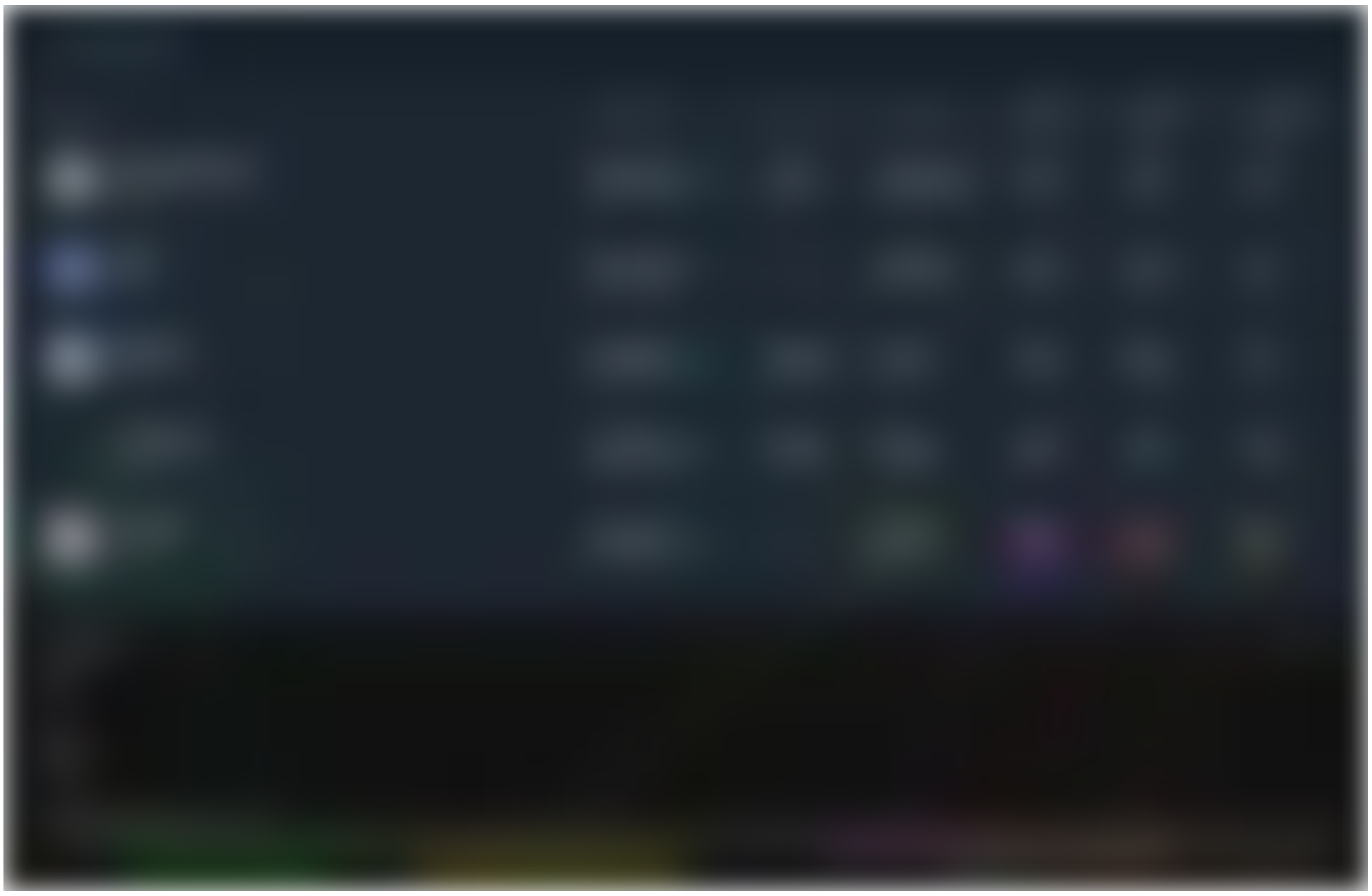
The documents and code do not tell us what the variable scale is used for, but it holds the number 1e18, so presumably it is to let the consumer know how to scale the percentages.

The meaning of these variables was explained in the article on liquidations and collateral.

When we query the current values for the token UNI (assetId 3) (https://etherscan.io/address/0xc3d688B66703497DAA19211EEdff47f25384cdc3#readProxyContract#F16), we can compare the values to the ones displayed on the marketplace. The relationship should be clear. Of note: the liquidation penalty is 1 - liquidation factor. The liquidateCollateralFactor is the LTV at which the loan gets liquidated. The liquidationFactor encodes the liquidation penalty. The fact that the liquidationFactor in the struct does not mean the same thing as the liquidationFactor in the UI is confusing.

The top image is the screenshot, and the values below is a screenshot from Etherscan querying getAssetInfo() for token UNI.

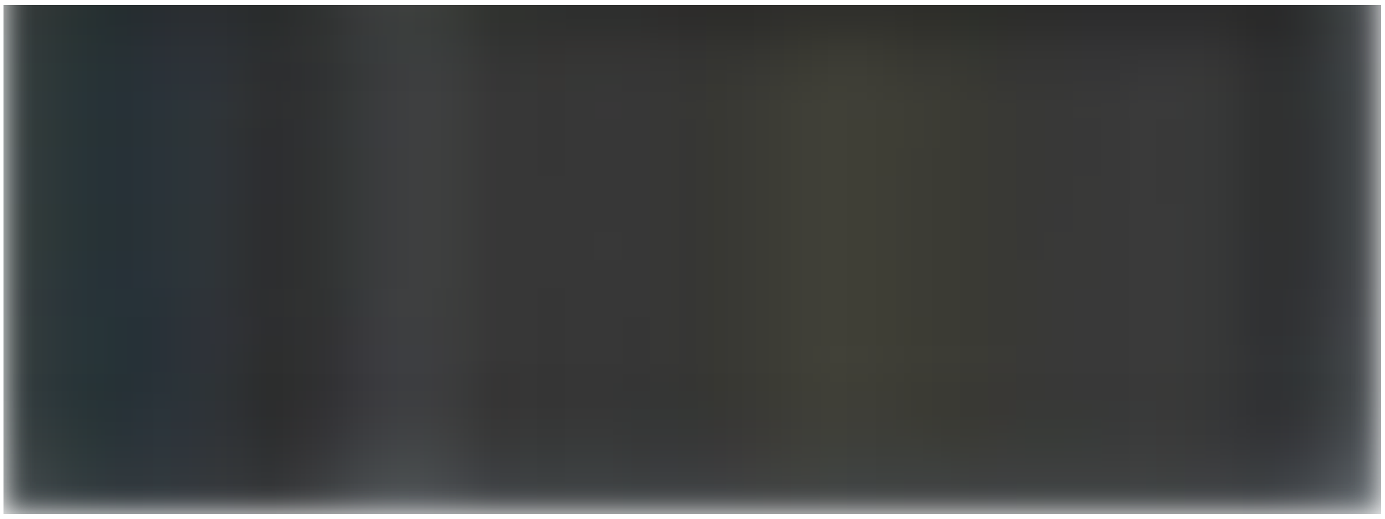
Below we show the relationship between the UNI collateral parameters on the Compound UI and Etherscan querying getAssetInfo() for token UNI.



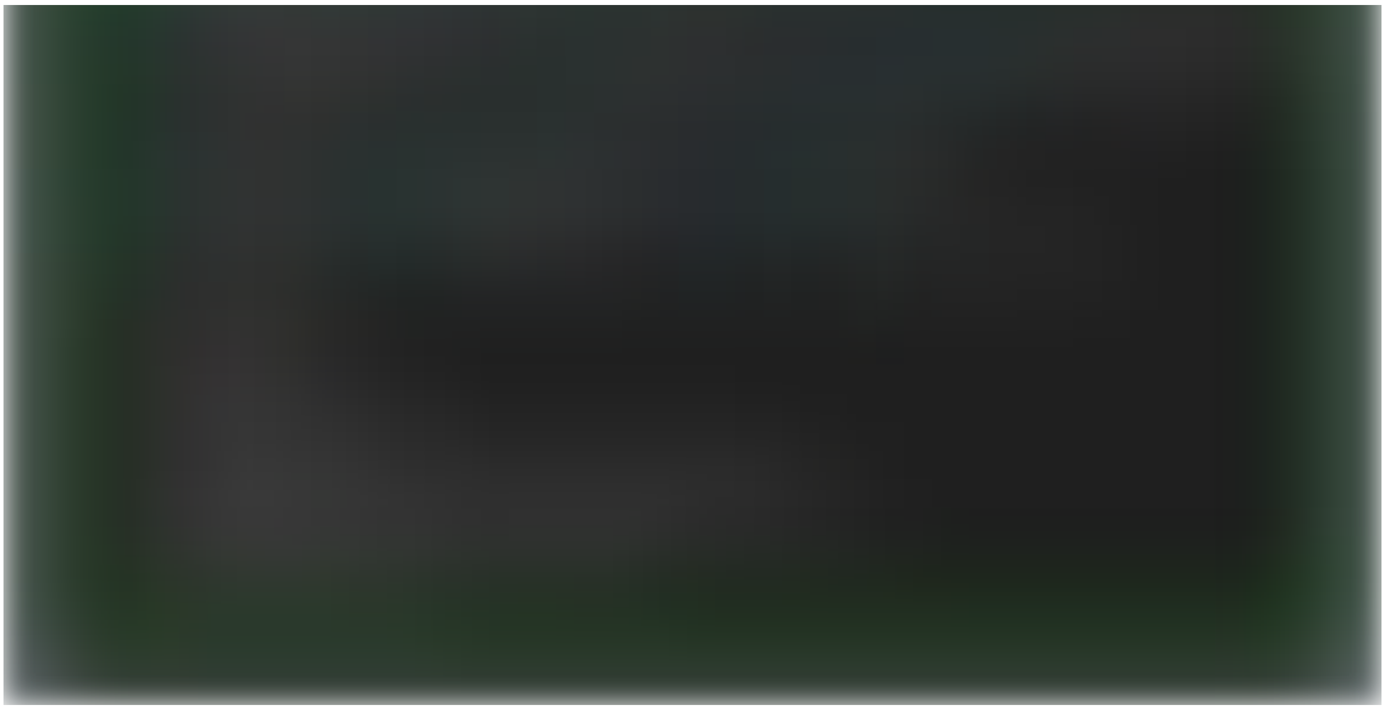
The obvious next question is, “where does Compound store the AssetInfo structs?”

AssetInfo “structs” are kept in immutable variables

Each asset’s info is packed into immutable variables — it is not kept in storage for gas efficiency purposes. Since it takes two 32 byte words to store the AssetInfo struct, Comet numbers off the uint256 words with assetXX_a, assetXX_b. The XX here indicates the asset index. So asset00_a and asset00_b collectively hold the AssetInfo struct for asset 0. Remember, it takes two 256 bit variables to store AssetInfo, which is 432 bits large.



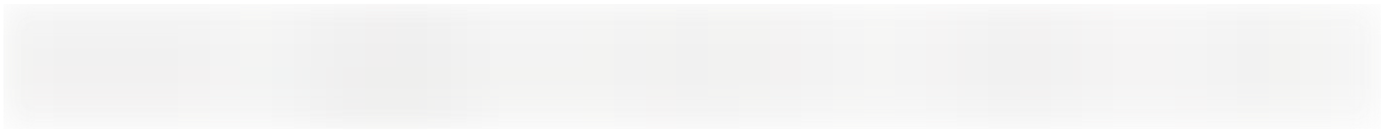
We can now show the implementation of getAssetInfo() from Comet.sol:280-356. It simply unpacks the immutable variable into the AccountInfo struct and returns it. The bitshifting and packing it uses is straightforward, so we will not explain it here.



Because these variables are immutable, governance must deploy a new implementation and update the proxy if it wishes to add another collateral asset or change the parameters of one of the assets. Care must be taken to only append assets and not interfere with previous definitions.

Checking if a borrower can be liquidated

The Comet.sol isLiquidatable() function sums up the collateral assets held by a user multiplied by their liquidationFactor. If this sum is less than the present value of their debt (which is a negative number), then the user is liquidatable.



This means a borrower might have one asset below the liquidation threshold, but if other collateral assets balance it out the deficit, then the user is not liquidateable.

The full value of the collateral does not “count” towards the user’s collateral balance — it is reduced by the liquidation factor.

Here is the same example from earlier showing the true value of the user’s collateral assets.

In the example above, the hypothetical borrower will get liquidated if their loan balance exceeds $8,360.

Absorbs are all or nothing — there is no option to liquidate part of the collateral in Compound V3. The entirety of the borrower's asset balances are set to zero.

Example absorb

Suppose Bob deposited $1000 of ETH into Compound V3 and borrowed $800 USDC. This satisfies the collateralization ratio of 80%. The value of ETH drops to $880 causing the LTV (loan to value) to hit 90.9%, triggering the 90% liquidation threshold.

A liquidator calls absorb() on Bob's account and the $880 of ETH collateral are absorbed into the protocol.

Let's say the liquidation penalty is 5%.

Since the collateral value is currently $880 ETH, 5% of that is $44 in ETH.

The protocol will deduct $44 from Bob's collateral as a penalty, leaving $836. Since Bob borrowed $800 USDC, there is a $36 surplus. That is, $800 is taken by the protocol to cover the debt leaving $36 left over. This is credited to Bob who now becomes a lender with a $36 USDC deposit.

Bob has already withdrawn the $800 USDC when he took out the loan, so his total holdings are now $836.

Note that nothing in the absorb() interaction directly rewarded the liquidator.

Note that **when a borrower gets liquidated, they will become a lender if the collateral is sufficient to cover the debt.**

If the collateral is not sufficient to cover the debt, then the protocol implicitly takes a loss from its *reserves* which we discuss next.

Reserves

Interest paid by borrowers that is in excess of what lenders earned are called "reserves" in Compound V3.

Reserves Example

Alice lends the protocol 100 USDC and earns 5% interest. Bob borrows 100 USDC from the protocol and pays 10% interest. For the sake of simplicity, let's assume Alice and Bob are the only actors in the system. There will be an extra 5% interest that Bob paid that Alice didn't earn. This extra amount is the reserve.

It doesn't matter whether Bob has paid back the loan or not (i.e. has he transferred 110 USDC to the protocol yet or not). He owes the protocol 110 USDC and the protocol owes Alice 105 USDC. Therefore, there are 5 USDC in reserves.

Suppose Bob pays off the loan. Now the protocol has a balance of 110 USDC, 105 of which is due to Alice. There are still 5 USDC in reserves — nothing changed.

The function getReserves() returns this value. The USDC the protcol "owns" is the sum of

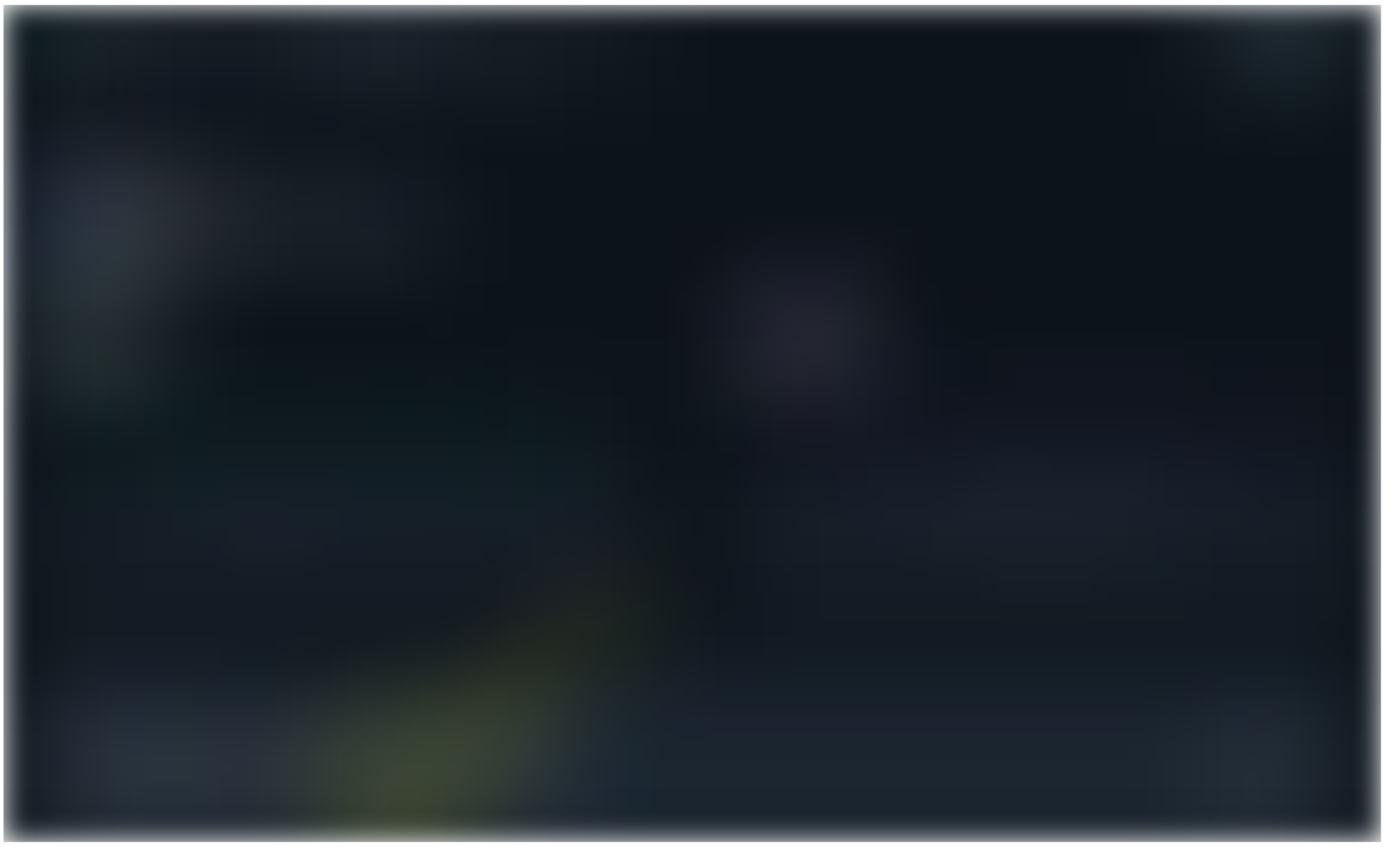
1) the balance of USDC held by Compound i.e. ERC20(baseToken).balanceOf(address(this)) and
2) the present value of the totalBorrow,
3) minus the amount the protocol owes to lenders, i.e. the present value of the totalSupply.

In other words, it is usdc_balance + totalBorrow - totalSupply.

As you can see in the function below, totalSupply is assigned a negative sign because that is what Compound owes to the lenders. The positive factors — the amount of held USDC and the net amount of USDC owed to Compound from borrowers — are the amount of USDC "owned" by the Compound.

If we look at the getReserves() function on Etherscan, we will see the reserves at the time of writing are 3.47 million USD (6 decimals).

When we look at the USDC / Mainnet Compound market, we also see the front-end displaying the current reserves as 3.47 million.



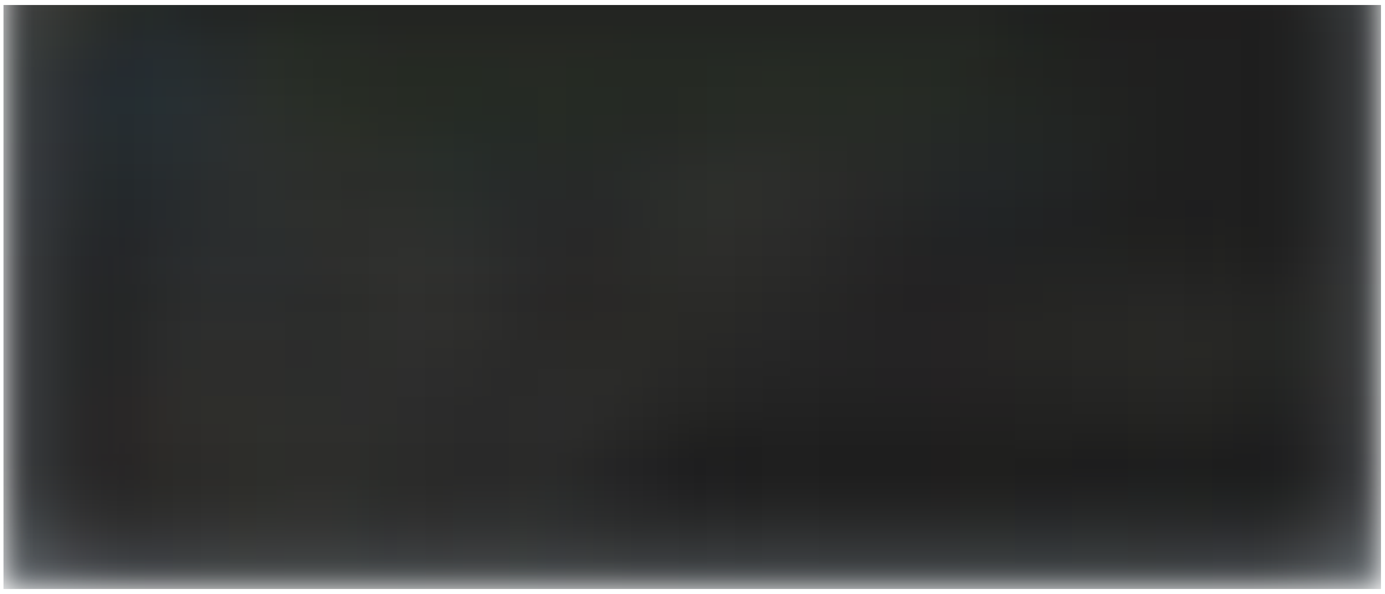
If we call getReserves() before and after an absorb() we would notice that the reserves decreased. This happens for two reasons:

1) The protocol paid off the loan (so less is owed to it). This came out of the reserves, so naturally there are less reserves.

2) The borrower becomes a lender with a small deposit. This deposit is owed by Compound to the lender, further decreasing the reserves.

withdrawReserves()

The excess reserve is for use by governance and can be withdrawn using the function below.



Target Reserves

Compound has a public immutable variable targetReserves defined in Comet.sol

When we look at the targetReserves on Etherscan, we see it is 5 million USDC.

Target reserves has only one use in the protocol: to determine if the protocol has enough "margin of safety" to not sell absorbed collateral. Note that governance could change this value by deploying a new Comet instance.

That is, **if Compound V3 has sufficient "excess cash," they'd rather hold on to the collateral for the purpose of speculating that it will increase in value.**

Let's examine the only function where this variable is used.

buyCollateral()

Collateral is still inside the protocol after an absorb. All that happened was that the user's collateral balance was set to zero — however the collateral was not transferred anywhere. This collateral is still "inside" Compound V3.

To incentivize liquidators, collateral held by Compound is sold at a discount via the buyCollateral() function.

There are two crucial pieces of business logic in this function:

1. If the reserves amount is larger than the target reserves ($5 million) this function will revert, not allowing liquidators to purchase collateral. (yellow box in the code below). As mentioned above, Compound wishes to speculate on the collateral. Since it is already in a cash-heavy position, they don't wish to accumulate more cash.

Recent Posts See All

Understanding the Function Selector in Solidity

12 0

How ERC721 Enumerable Works

172 0 2

To liquidate a borrower, the liquidator calls absorb() with the borrower's account as arguments, then calls buyCollateral() in the same transaction. The liquidator should check that reserves are not more than the target reserves, and that the account is liquidate able

Subscribe to our newsletter Subscribe Now

We do not sell your information to anyone. Period.

Web3 Blockchain Bootcamp