GHOOK Hackaton Report

Optimizing Liquidity Providing through Decentralized Borrow Protocols.

Abstract:
The decentralized finance (DeFi) ecosystem has seen tremendous growth in recent years, of dollars in total value locked (TVL) in various protocols. This growth in TVL can be largely associated with the rise of Decentralized Exchanges (DEX), though entities that deposit their liquidity tokens into Liquidity Pools, also known as Liquidity Providers. However, it is estimated that up to 50% of liquidity providers provide liquidity at a loss through Impermanent Loss (IL). The goal of this project is to allow more flexibility for Liquidity Providers, by allowing them to borrow assets against their liquidity position though overcollateralized lending/borrowing. It consists in an implementation of a lending and borrowing protocol through the use of Uniswap's v4 hooks, and AAVE's stablecoin GHO, two of DeFi's biggest protocols.

Keywords:
Blockchain, Decentralized Finance (DeFi) ,Decentralized Exchange (DEX), Volatility Analysis, Overcollateralized Borrowing

Acronyms and abbreviations

DEFI: Decentralized Finance.
DEX: Decentralized Exchange
CEX: Centralized Exchange
LP: Liquidity Pool token.
TVL: Total Value Locked.
IV: Implied Volatility
GHO: AAVE’s stablecoin

Table of Contents:

1: Introduction

Decentralized finance (DeFi) is a financial system that built on top of decentralized networks such as blockchain. DeFi aims to provide financial services such as lending, borrowing, and trading without the need for central intermediaries such as banks.

One of the main aspects of DeFI that differs the most from traditional finance is the concept of Liquidity Providing. Liquidity is what enables trades, since a platform must have enough liquidity to enable traders to buy or sell assets. The way liquidity is provided in DeFi most of the time is through the use of Decentralized Exchanges, the biggest being currently Uniswap.

Decentralized Exchanges bring a new paradigm to trading through the use of Liquidity Pools and Liquidity Providers. In Decentralized Exchanges, there is often no order book, but rather a pool of two assets (a quote and a base asset), that anybody can provide in different quantities (see Figure 1). Traders can then swap back and forth the provided tokens from the Liquidity Providers whenever they want in a decentralized way.

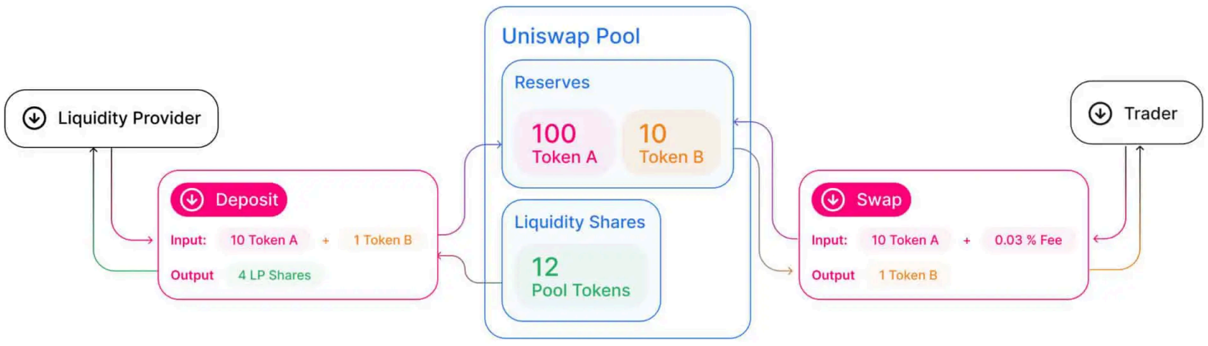

Figure 1: A Uniswap Pool Liquidity Flow

Liquidity Providers are incentivized to deposit through the earnings they get whenever somebody trades assets in their pool. They earn fees on every swap, proportional to their percentage of the pool total amount. Their liquidity is locked inside the pool, and as long as it stays in the Pool, they continue earning Fees from the swaps.

In traditional finance, it is mostly achieved through order books and market makers that have very big amounts of money (i.e liquidity) to facilitate trading. These market makers are often big platforms, such as Binance, Coinbase, or OKX. Uniswap and other decentralized exchanges are called Automated Market Makers (AMM) since they automatically use liquidity from the Liquidity Providers to enable trading, and the Pool Liquidity is made up of several thousands of Liquidity Providers that don’t know each other.

One of the main problems Liquidity Providers face is that their liquidity is blocked inside the pool. While it provides passive earning though swap fees, it lacks capital efficiency for

Liquidity Providers. Some protocols, such as Silo, are trying to mitigate that by allowing LPs to borrow stablecoins against their LP tokens (for Silo, it is Curve LP tokens). This project aims to allow Uniswap v4 LPs to borrow GHO (AAVE's stablecoin, explained in the Background Section) against their Liquidity Position.

While providing liquidity is a great way to earn passive income on your cryptocurrencies, most liquidity providers (LP) are not profitable from providing liquidity in Liquidity Pools. In fact, according to this source 50% of Liquidity Providers Lose Money.

2: Background

Uniswap is one of the biggest and most prolific DEX in EVM chains. Uniswap had several versions over the course of the years. We won't cover uniswap v1 and v2 in this paper. However Uniswap v3 and v4 are worth covering as the implementation of this project was made using Uniswap v4.

Uniswap v3 differs from previous iterations with the introduction of a concentrated liquidity concept. Liquidity Providers now have the ability to supply their assets in a definite price range for which they deposit liquidity.
With Uniswap v3 concentrated liquidity, liquidity providers can possibly gain higher returns on their capital with as much as 4000x efficiency. While Uniswap v3 provides greater returns for LPs, it also increases Impermanent Loss the more concentrated the liquidity is.

Impermanent Loss occurs when LPs deposit into a liquidity pool and the price of the tokens change. The larger the change in price of the tokens compared to when they were deposited the larger the loss. This is explained by the fact that when somebody trades a crypto, for instance by buying it from a pool, he effectively takes away Asset A (the one he is buying) from the pool, while depositing Asset B (the one he is selling) inside the pool. The pool ratio between Asset A and Asset B is then disequilibrated, which is what is called Impermanent Loss. Here is an example :

- Luke is a liquidity provider who wishes to commit $200 to a liquidity pool that contains ETH/USDC pair.

- This is a 50/50 pool which means that both tokens in the pair must be equivalent in value.

- If 1 ETH is $100 and 1 USDC is $1 therefore 1 ETH is 100 USDC. Luke will commit his $200 to the pool by providing 1 ETH and 100 USDC.

- Now there is a total of 10 ETH/10,000 USDC in the pool deposited by other LPs. Luke is therefore entitled to 10% of the liquidity pool.

- After Luke commits his $200 the price of ETH rallied to $400. This creates a discrepancy in the price of ETH in the pool and the price of ETH in the market (other exchanges)since ETH in the pool is cheaper than ETH in the market.

- This creates an arbitrage opportunity, arbitrage traders will add USDC to the pool to get ETH until the ratio reflects the current price of ETH.

- Now the ratio between ETH/USDC in the pool has changed and there is now 5 ETH and 20,000 USDC in the pool.

- If Luke withdraws his 10% from the pool he will now get 0.5 ETH and 200 USDC which gives him $400

- He made a profit right? Well if Luke had just simply used $100 to buy 1 ETH and buy 100 USDC and hold both assets without providing liquidity, he would have now had 1 ETH * 400$ + 100$ = $500 in total.

- So this difference of $100 is the impermanent loss. While technically the loss becomes permanent if Luke exits the pool, if he does not withdraw his deposit from the pool, there is a chance that the price of ETH will return back to 100 USDC. If Luke withdraws his liquidity when the price is back at the same price, he effectively won't lose anything, and make profit from the swaps made in the pool.

Uniswap v4 is a new version of Uniswap, that still has concentrated liquidity, and brings, with many other features, the concept of "Hooks".In Uniswap V4, hooks are special pieces of code that add custom features to a liquidity pool. When you create a pool, you can attach a "hook contract" that spells out specific actions to occur at certain times during a transaction. These hooks operate as optional add-ons and can be set up to manage swap fees or in our case allow to mint AAVE's decentralized stablecoin GHO.

GHO is a decentralized, overcollateralized stablecoin native to the Aave protocol, the biggest lending / borrowing protocol of DeFI. This means that the token is initially minted from crypto assets supplied to the Aave protocol. Additionally, the token lives on the Ethereum blockchain, just like Aave, and it follows the ERC-20 token standard.

The GHO token is programmed to be aligned with USD via market efficiency. So, despite market volatility, GHO should always stay pegged to the US dollar (with minor deviations).

GHO introduces the concept of facilitators. It points to entities that can deploy different strategies to mint and burn the GHO cryptocurrency trustlessly. Further, the Aave DAO assigns each facilitator a specific upward limit of how much GHO the entity can mint. This limit is known as a "bucket". Minting GHO is made through AAVE's GHO smart contracts.

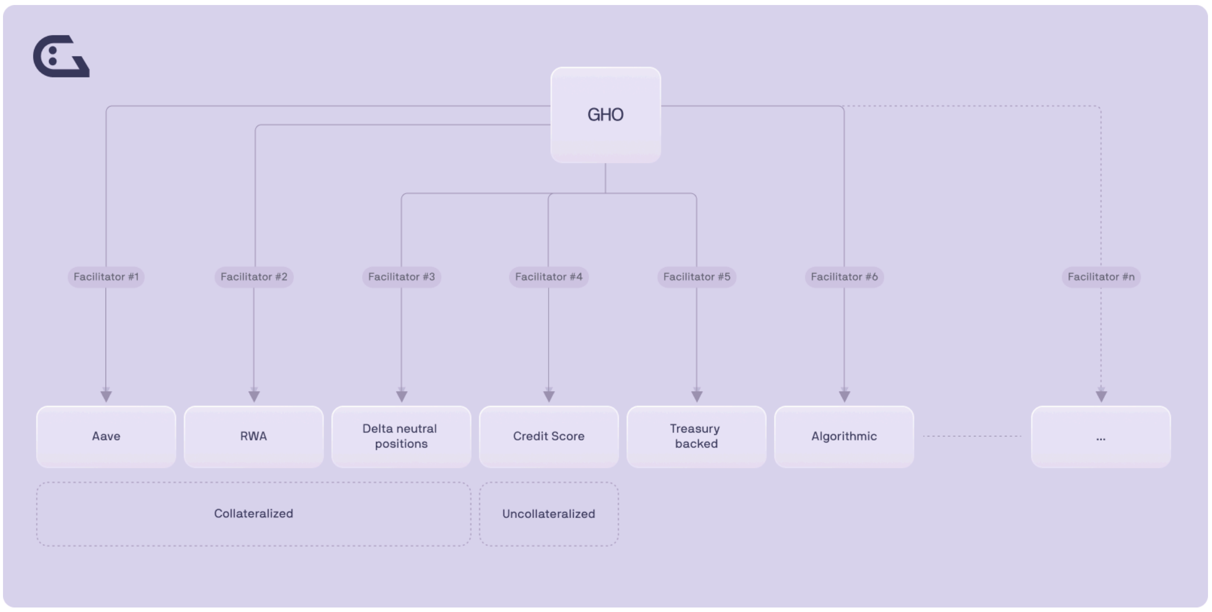

Figure 3: GHO facilitators hierarchy

3. Liquidity Providers and added flexibility of borrowing

Allowing Liquidity Providers to borrow against their Liquidity Position brings a few advantages. First, Liquidity is not locked inside the pool, and we can use the borrowed liquidity for other purposes. These purposes include:

-Liquidity Farming: Borrowed Liquidity can be used elsewhere to generate yield, for instance, providing USDC on AAVE yields around 5% yearly, and can be used as another source of yield on top of swap fees.

-Leveraged Liquidity farming: Borrowed Liquidity can be used to put more tokens back into the Liquidity Position, then borrow more and make loops. Liquidity Providers then gain more exposure to the price movement of the Liquidity Position,while earning more fees from their leveraged position but also have more debt.

-Hedging: Borrowed Liquidity can be used to buy options or perpetuals to hedge a liquidity position. If Liquidity providers anticipate a decrease in price, they can buy put options or short perpetuals to cover their loss from the Liquidity Position. Conversely, if Liquidity Providers anticipate an increase in price, they can buy a call or a long perpetual to cover the loss from their LP that will progressively convert to the quote asset.

Borrowing against a Liquidity Position allows for overall greater capital efficiency, while increasing flexibility for Liquidity Providers since their Liquidity is not fully locked anymore, but can be partially unlocked.

The main problem with this type of approach is that if we want Liquidity Providers to borrow, we need lenders to provide the said assets to borrow. For instance, if a Liquidity Provider has a position worth 1000$, we need a lender to be able to provide him say 700$. Therefore for every Borrower we need Lenders, and it is difficult to attract Lenders since we need to pay them sometimes very high lending rates (AAVE’s USDC borrow rates sometimes reach 30% in case of high utilization).

This project solves this problem by using GHO. GHO, the decentralized stablecoin by AAVE, can be freely minted by any whitelisted entity, provided there is enough collateral to back it. Furthermore, GHO borrow rates are fixed, and are of 4,49% with stkAAVE discount, and 6,42% without discount. It solves both problems of having the need of lenders since with GHO there is no need for lenders, because GHO can be freely minted against a Liquidity Position, and burned when the debt is repaid. It also has fixed borrow rates that don’t vary on utilization rates, so Liquidity Providers can know exactly how much they will have to pay back after a certain time.

Combining the programmability of Uniswap v4 hooks with GHO design allow to make an efficient borrowing protocol, with many upsides versus traditional lending / borrowing protocols. It offers flexible borrowing conditions for Liquidity Providers while being very capital efficient.

5. Implementation using Uniswap v4 modular “hook” framework and AAVE’s GHO Stablecoin

Uniswap v4 introduces the concept of “hooks”. Hooks are arbitrary functions that can be executed at specific moments of the pool life cycle, for instance before or after a swap, or before / after providing liquidity.

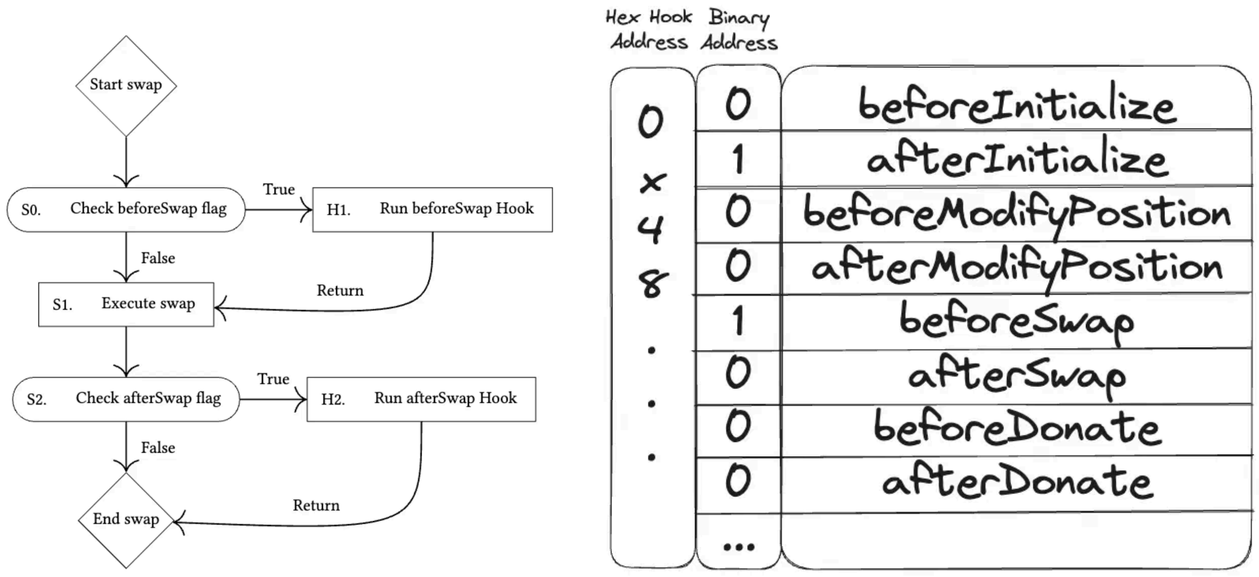

Figure 5: Unisawp v4 hooks flags

Our goal in this project is to implement functions that allow a Liquidity Provider to mint GHO trustlessly from the hook contract, while checking he has enough collateral, and maintain a Loan to Value inferior to the Max Loan to Value calculated in part 3.

The implementation in this project is made using a template for making hooks on uniswap v4. Please note this specific repository does not allow for liquidation (i.e: sell the position if there is more debt than collateral) since by design, Uniswap v4 does not allow an external operator to modify the position of someone (see later the LockAcquired function). This difficulty has been overcome using Bungi’s LP position Manager, by making a contract that manages position in the name of the user.

The function **beforeModifyPosition** is part of the overridable functions of uniswap v4 hooks, and takes as parameter the address of the owner of the position, a PoolKey that contains information about the two currencies inside the pool and the hook address, and the ModifyPositionParams that contains the parameter of the position.

```
/// @inheritdoc IHooks

    function beforeModifyPosition(

        address owner, // sender

        IPoolManager.PoolKey calldata, // key

        IPoolManager.ModifyPositionParams calldata params// params

    )

        external

        override

        returns (bytes4)

    {

      if(params.liquidityDelta < 0 ){

            //If user try to withdraw (delta negative) and has debt, revert

            uint256 liquidity = uint256(-params.liquidityDelta);

            console2.log("liquidity to withdraw %e", uint128(liquidity));

            if(!_canUserWithdraw(owner, params.tickLower, params.tickUpper,
uint128(liquidity))){

                 revert("Cannot Withdraw because LTV is inferior to min LTV");

            }

        }

        console2.log("beforeModifyPosition");

        return IHooks.beforeModifyPosition.selector;

    }
```

Note that when the **params.liquidityDelta** is inferior to 0, it indicates that the Liquidity Provider removes the specified amount of liquidity. We therefore have a function _canUserWithdraw that checks if the liquidity can be removed based on the debt of the Liquidity Provider. If the value of the collateral after withdrawal is inferior to the debt, adjusted to the max LTV, then we revert since there cannot be more debt than collateral.

The function **borrowGho** allows a user with an existing Liquidity Position to borrow GHO based on the value of his Liquidity Position. If the user tries to borrow more GHO than the value of his position, adjusted to the max LTV, it will revert

```
function borrowGho(uint256 amount, address user) public returns (bool,
uint256){
        //if amount is inferior to min amount, revert
        if(amount < minBorrowAmount){
            revert("Borrow amount to borrow is inferior to 1 GHO");
        }

        console2.log("Borrow amount requested %e", amount);
        console2.log("Max borrow amount %e",
_getUserLiquidityPriceUSD(user).sub((UD60x18.wrap(usersDebt.get(user))).div(UD
60x18.wrap(10**ERC20(gho).decimals()))).mul(maxLTVUD60x18).unwrap());
        //get user position price in USD, then check if borrow amount + debt
already owed (adjusted to gho decimals) is inferior to maxLTV (80% =
maxLTV/100)
        if(_getUserLiquidityPriceUSD(user).lte((UD60x18.wrap((amount+
usersDebt.get(user))).div(UD60x18.wrap(10**ERC20(gho).decimals()))).mul(maxLTV
UD60x18))){
            revert("user LTV is superior to maximum LTV"); //TODO add proper
error message
        }
        usersDebt.set(user, usersDebt.get(user) + amount);
        IGhoToken(gho).mint(user, amount);
    }
```

Note: We use the **UD60x18** from the PRB-Math library because it provides gas efficient decimal representation of numbers in solidity, which is used in many parts of this project to compute the price of the collateral.

The function **repayGho** allows a user that borrowed using the **borrowGho function** to repay his debt. When repaying his debt, we burn the repaid GHO, which effectively reduces the available supply of GHO, helping it keep its peg around 1$.

```
function repayGho(uint256 amount, address user) public returns (bool){
        //check if user has debt already
        if(usersDebt.get(user) < amount){
            revert("user debt is inferior to amount to repay");
        }
        //check if user has enough gho to repay, need to approve first then
repay
        bool isSuccess = ERC20(gho).transferFrom(user, address(this), amount);
//send gho to this address then burning it
        if(!isSuccess){
            revert("transferFrom failed");
            return false;
        }else{
            IGhoToken(gho).burn(amount);
            usersDebt.set(user, usersDebt.get(user) - amount);
            return true;
        }

    }
```

We have the internal function **_getPositionUsdPrice** which takes a position parameters (tickLower, tickUpper and liquidity), and calculates the price is US dollar of the position. This function uses the range math from Uniswap v3. Once we have the amount of tokenA and tokenB, we multiply their amount by their price using ChainLink Price feed, to avoid possible price manipulation from the internal price of the pool.

```
function _getPositionUsdPrice(int24 tickLower, int24 tickUpper, uint128
liquidity) internal view returns (UD60x18){
        IPoolManager.PoolKey memory key = _getPoolKey();
        (uint160 sqrtPriceX96, int24 currentTick, , , , ) =
poolManager.getSlot0(key.toId()); //curent price and tick of the pool

        //Lower and Upper tick of the position
        uint160 sqrtPriceLower = TickMath.getSqrtRatioAtTick(tickLower); //get
price as decimal from Q64.96 format
        uint160 sqrtPriceUpper = TickMath.getSqrtRatioAtTick(tickUpper);
        uint256 token0amount;
        uint256 token1amount;
        //Price calculations on
https://blog.uniswap.org/uniswap-v3-math-primer-2#how-to-calculate-current-hol
dings
        //Out of range, on the downside
        if(currentTick < tickLower){
            token0amount = SqrtPriceMath.getAmount0Delta(
                sqrtPriceLower,
                sqrtPriceUpper,
                liquidity,
                false
            );
            token1amount = 0;
        //Out of range, on the upside
        }else if(currentTick >= tickUpper){
            token0amount = 0;
            token1amount = SqrtPriceMath.getAmount1Delta(
                sqrtPriceLower,
                sqrtPriceUpper,
                liquidity,
                false
            );
        //in range position
        }else{
            token0amount = SqrtPriceMath.getAmount0Delta(
                sqrtPriceX96,
                sqrtPriceUpper,
                liquidity,
                false
            );
            token1amount = SqrtPriceMath.getAmount1Delta(
                sqrtPriceLower,
                sqrtPriceX96,
                liquidity,
                false
            );
        }

        //Use UD60x18 to convert token amount to decimal adjusted to avoid
overflow errors
        UD60x18 token0amountUD60x18 =
UD60x18.wrap(token0amount).div(UD60x18.wrap(10**ERC20(Currency.unwrap(key.curr
ency0)).decimals()));
        UD60x18 token1amountUD60x18 =
UD60x18.wrap(token1amount).div(UD60x18.wrap(10**ERC20(Currency.unwrap(key.curr
ency1)).decimals()));

        //Price feed from Chainlink, convert to UD60x18 to avoid overflow
errors
        UD60x18 ETHPrice =
UD60x18.wrap(uint256(ETHPriceFeed.latestAnswer())).div(UD60x18.wrap(10**ETHPri
ceFeed.decimals()));
        UD60x18 USDCPrice =
UD60x18.wrap(uint256(USDCPriceFeed.latestAnswer())).div(UD60x18.wrap(10**USDCP
riceFeed.decimals()));

        //Price value of each token in the position
        UD60x18 token0Price = token0amountUD60x18.mul(ETHPrice);
        UD60x18 token1Price = token1amountUD60x18.mul(USDCPrice);

        //return price value of the position as UD60x18
        return token0Price.add(token1Price);

    }
```

We have the function **liquidateUser**. Liquidation occurs when the value of a Liquidity Position goes under the max LTV. In that case, anybody can repay the borrower's debt. Then, we withdraw the LP position from the pool, and distribute the collateral tokens, partially back to the original LP, and partially to the liquidator (i.e: the person that called the function, most of the time bots) as a liquidation premium.

```
/// @notice Given an existing position, liquidate position by repaying debt,
then withdrawing collateral
    ///     This function supports partially withdrawing tokens from an LP to
open up a new position
    /// @param owner The owner of the position
    /// @param position The position to liquidate
    /// @param hookLiquidationData the arbitrary bytes to provide to hooks
when the existing position is modified
    function liquidateUser(
        address owner,
        Position memory position,
        bytes calldata hookLiquidationData
    ) external returns (bool liquidationSuccess) {

        if(getUserCurrentLTV(owner) < maxLTVUD60x18){
            revert("User LTV is not at risk of liquidation");
        }

        uint8 liquidationPremium = 20; //20% of GHO debt to liquidator

        //get user Current Position and debt
        BorrowerPosition storage currentParams = userPosition[owner];

        //send GHO to this address then burning it
        bool isTransferSuccess = ERC20(GHO).transferFrom(msg.sender,
address(this), currentParams.debt);

        if(!isTransferSuccess){
            revert("GHO transferFrom failed");
        }

        //burn GHO debt
        IGhoToken(GHO).burn(currentParams.debt);

        //reset user debt to 0
        userPosition[owner].debt = 0;
        //burn ERC6909 position tokens
        _burn(owner, currentParams.position.toTokenId(),
uint256(currentParams.liquidity));


        //Set Position params to 0 to liquidate
        IPoolManager.ModifyPositionParams memory liquidationParams =
IPoolManager.ModifyPositionParams({
            tickLower: currentParams.position.tickLower,
            tickUpper: currentParams.position.tickUpper,
            liquidityDelta: -int256(int128(currentParams.liquidity))
        });

       uint256 token0balance = ERC20(WETH).balanceOf(address(this));
       uint256 token1balance = ERC20(USDC).balanceOf(address(this));

        // interactions, second parameter is receiver of tokens.
        BalanceDelta delta = abi.decode(
            manager.lock(
                abi.encodeCall(
                    this.handleModifyPosition,
abi.encode(CallbackData(msg.sender, address(this), poolKey, liquidationParams,
hookLiquidationData))
                )
            ),
            (BalanceDelta)
        );

        //After the call, balances should be settled and we should receive
positions tokens back here.
        token0balance = ERC20(WETH).balanceOf(address(this)) - token0balance;
//get actual received token0 amount after withdrawing position
        token1balance = ERC20(USDC).balanceOf(address(this)) - token1balance;
//get actual received token1 amount after withdrawing position

        console2.log("ETH balance after actual liquidation %e",
token0balance);
        console2.log("USDC balance after actual liquidation %e",
token1balance);

        IERC20(WETH).transferFrom(address(this), msg.sender,
(token0balance*liquidationPremium)/100); //send 20% ETH to liquidator as
liquidation premium
        IERC20(USDC).transferFrom(address(this), msg.sender,
(token1balance*liquidationPremium)/100); //send 20% USDc to liquidator as
liquidation premium

IERC20(WETH).transferFrom(address(this),address(owner),(token0balance*(100-liq
uidationPremium)/100)); //send 80% ETH to original user

IERC20(USDC).transferFrom(address(this),address(owner),(token1balance*(100-liq
uidationPremium)/100)); //send 80% USDC to original user

        return(userPosition[owner].debt == 0);
    }
```

What prevents us from doing a liquidate function on the original implementation is the **onlyLocker** modifier.

```
 modifier onlyByLocker() {
        address locker = lockData.getActiveLock();
        if (msg.sender != locker) revert LockedBy(locker);
        _;
    }
```

This modifier prevents anybody that's not the locker, in our case when we add liquidity to the pool, it is the owner of the position, to act on the position of the locker.

The Liquidity Manager repository implements a **LiquidityPositionManager** contract that handles the operation such as adding or removing liquidity in the name of the owner. This way, we can remove the liquidity from the Position Manager in case of a liquidation.

Without Uniswap v4 allowing the hook itself to modify the position of users, an implementation without a Liquidity Manager seems impossible to do.

6. Limitations

Due to their dual token nature, Liquidity Positions have more underlying risk than single assets. First, Liquidity Positions always take the "worst side" of the trade because of

Impermanent Loss. If for instance we have an ETH/USDC LP, then if the price of ETH goes up, the LP underlying assets will shift towards USDC, thus making the LP less profitable that holding pure ETH, and providing it to a lending protocol such as AAVE. If the price of ETH goes down, the LP shifts more towards ETH, although the LP loses less value than pure ETH. One may think of Uniswap LPs as a synthetic asset that grows with the square root of the price of ETH as long as it stays in range. When going outside the range, the LP acts the same as the underlying asset, i.e USDC on the upside if ETH price goes above the maximum tick of the position, and ETH if the price is on the downside of the minimum tick of the position.

There is also the potential of price manipulation, where an attacker uses a flashloan (i.e borrowing huge amounts of token, then repaying them in a single transaction) to make the price of the pool go way down, triggering liquidations for users, and taking the liquidation premium for himself. This type of attack is somewhat limited by using the price oracles from ChainLink, but the calculations of the price tick still rely on the current price from the pool.

Another limitation is the risk of depeg from GHO, i.e the variation away from 1$ from the GHO stablecoin.At the time of writing this publication, GHO has been trading around 0.98$, with a lowest price of 0.95$ at some point. It may provide great incentive to repay the GHO debt for the borrowers, since their debt may now be “cheaper”, but provide greater risk for people that still have exposure to GHO by holding it.

7. Conclusion

Allowing Liquidity Providers to borrow against their liquidation position provides greater capital efficiency and flexibility for borrowers. While traditional lending protocols need lenders to provide liquidity that can be borrowed, GHO solves this problem by allowing to trustlessly mint it when backed by enough collateral. With this approach, we do not need lenders to provide stablecoins to be borrowed, and we can provide as much GHO stablecoin as needed for borrowers. When a borrower's position value is under 80% of the value borrowed, a liquidation happens, meaning the borrower's position is sold to repay the debt. This allows the system to remain operational, and ensures GHO is always be backed by at least 1$ worth of collateral.

This approach is beneficial to Liquidity Providers as it allows greater capital efficiency, while adding flexibility for the Liquidity Provider. Value from the Liquidity Providers can be freely extracted from their position, allowing them to use it for other purposes such as yield farming, leveraged Liquidity Providing, and hedging their position.

To implement this approach, we used Uniswap v4 modular “hook” approach, which allows us to implement arbitrary functions for given actions on the pool. They allow to mint GHO from AAVE's contracts if there is enough collateral backing it.

References

[1]: Austin Adams, Sara Reynolds, Kirill Naumov, and Rachel Eichenberger. “A Primer on Uniswap v3 Math Part 2: Stay Awake by Reading it Aloud”, blog.uniswap.org

[2]: Guillaume Lambert.“On-chain Volatility and Uniswap v3”, lambert-guillaume.medium.com

[3]: Guillaume Lambert. “Pricing Uniswap v3 LP Positions: Towards a New Options Paradigm?” , lambert-guillaume.medium.com

[4]: Guillaume Lambert. “Understanding the Value of Uniswap v3 Liquidity Positions” , lambert-guillaume.medium.com

[5]: Emilio Frangella, Steven Valeri. “GHO,A flexible, decentralized stablecoin”, github.com/aave/gho-core

[6]: Hayden Adams, Moody Salem, Noah Zinsmeister, Sara Reynolds, Austin Adams, Will Pote, Mark Toda, Alice Henshaw, Emily Williams, Dan Robinson. “Uniswap v4 Core”, github.com/Uniswap/v4-core