Lending Base Layer

team@multiplier.tech

July 2024

DRAFT

Abstract

Lending Base Layer is a decentralised protocol designed to serve as universal infrastructure for collateral-based financial products. The protocol brings together mechanical elements common to a wide range of such designs, without imposing restrictions on their commercial logic, such as capital structure, risk management and pricing. It comprises a dual borrower-lender accounting system, a mechanism for entering/exiting an abstract borrowing relationship paired with transfer of authority over collateral, and an intents-based mechanism for state transitions.

1 INTRODUCTION

Blockchain-based lending protocols have found strong product-market fit from the early days of decentralised finance (DeFi). One of their key success factors has arguably been relative simplicity – foregoing feature richness and capital efficiency in favour of lower execution risk. However, with increased maturity and understanding of such mechanisms, this trade-off is becoming less favourable. Traditional finance provides plentiful evidence that serving the full spectrum of demand requires a varied and sophisticated product range, which DeFi needs to deliver in order to compete on global scale.

A smart contract system designed for the Ethereum Virtual Machine, Lending Base Layer seeks to simplify the process of building more complex and performant solutions by providing universal lending "back office" that support a variety of commercial terms. The protocol provides a system of accounts capable of expressing arbitrary borrower-lender relationships, a mechanism for automatic transfer of authority over collateral upon borrowing/repayment of debt, and a state transition mechanism that natively supports intents-based logic. The main goal of this infrastructure is to provide product designers with off-the-shelf mechanisms covering key elements of most lending products, allowing them to focus on commercial aspects of their offering.

The protocol's architecture enables independent parties to offer an unlimited number of risk-segregated products in a unified environment, to freely define their degree of interconnectedness and rules for their individual and aggregated balance sheets. The following sections expand on key elements of the mechanism and provide some intuition for their use in practice.

2 PROTOCOL OVERVIEW

2.1 High-Level Architecture

Lending Base Layer is a unified, smart contract-based synthetic environment featuring a system of borrower and lender accounts that brings together regular token balances with variables that define lending relationships.

A key part of the protocol's architecture is the concept of authority. Defined more precisely in sections that follow, it can be viewed as ability to exercise control over state of an account at a given time. In collateral-based lending, lenders always seek a degree of control over assets that borrowers provide as security. In the extreme, a lender can take direct possession of those assets for the duration of the loan, but that is not always necessary or desirable. Under the present design, a transfer of authority eliminates the need to move borrower's assets. Instead, the lender is able to specify arbitrary rules governing what the borrower can do with the contents of their account for the duration of the loan. Known to the borrower in advance, these rules can leverage the protocol's intents-based architecture to define permitted states at the end of each transaction (e.g. maximum loan-to-value), without placing any action-based restrictions on the borrower.

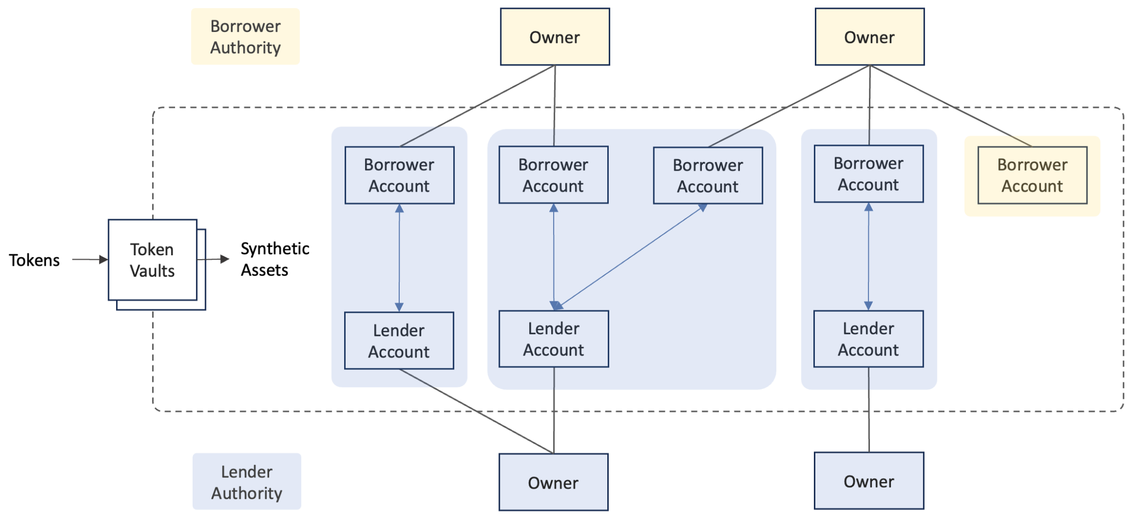

Figure 1: Lending Base Layer Architecture

2.2 Token Vaults

The protocol imports external value through a system of vaults. Each vault holds a single external token balance and issues corresponding one-to-one synthetic assets, which then circulate freely in the base layer environment. For a given vault V:

(1) $Vault_V = \{TokenBalance_V, TotalSyntheticBalance_V\}$

Vaults can be freely deployed by any user, and are configurable in two ways – through choice of vault builder pattern and controller. Vault builder patterns contain logic for interacting with a chosen token standard (e.g. ERC20). Templates are provided for most of the common ERC standards from the outset, and new ones can be created in the future as required.

The optional vault controller is intended to cater to use cases where underlying tokens carry an increased regulatory burden (e.g. ‘real-world assets”), providing vault creators with requisite levers to fulfill their obligations. In contrast, a vault that is built using a standard pattern template and has no controller is completely decentralised.

2.3 Accounts

The protocol features two distinct types of accounts – for borrowers and lenders. Both have an external *Owner* and contain *SyntheticBalance* representing a claim on tokens contained in vaults. They also feature abstract *BorrowBalance* and *LendBalance*, quantifying borrower-lender relationships in which they are engaged. The former means that borrower B owes lender L a number of units of debt, denominated in synthetic asset issued by vault V. In the same way, the latter means that lender L is owed by borrower B. These variables are effectively mirror-images of each other, defined separately to ensure that borrowing is done with the consent of both parties.

For borrowers B and lenders L:

(2) $Account_B = \{(SyntheticBalance_V), (BorrowBalance_{V,L,B}), Owner, Terms\}$

(3) $Account_L = \{(SyntheticBalance_V), (LendBalance_{V,L,B}), Owner, Terms\}$

Terms allow all accounts to support arbitrary smart contract logic, providing owners with a mechanism to delegate some of their authority over the account. In practice, it allows lenders to implement commercial functionality, such as permitting borrowers to decrement lender’s synthetic balance together with incrementing borrow and lend balances, which together constitutes borrowing. Both account types could potentially use this mechanism for a wide variety of purposes, such as enabling limit orders or implementing an automated market maker over account variables.

By design, borrowers cannot lend and lenders cannot borrow. Furthermore, the protocol prevents borrowers from having more than one lender, to ensure uniqueness of authority over collateral.

2.4 Batch

Batch is the sole mechanism for modifying base layer state. Through its intents-centric architecture, it enables monolithic execution of an arbitrary sequence of methods subject to checks on the end state. This is achieved by dividing the batch into two phases – execution and verification. The former can include logic that changes state variables, and also contains a mechanism to request additional logic to be run in the verification phase (*requestCheck*), where base layer state can no longer be modified but checks can be performed. The following configuration may offer a useful mental model for a typical batch:

(4) [**execution**: $Method_0, Method_1$ ... | **verification**: $Check_0, Check_1$ ...]

There are no restrictions on the nature of logic that can be executed in a batch, but modifying base layer variables can only be done through native methods detailed in sections that follow.

2.5 Authority

The protocol implements the notion of collateral through transfer of authority over borrower's account to the lender, removing the need to move underlying assets. Since borrowers can create unlimited accounts, they can restrict each account's contents to assets they specifically intend to post as collateral, thus making the mechanism economically equivalent to depositing assets directly with a lender, but mechanically simpler.

To achieve this functionality, we construct an authority check that must be passed in any attempt to modify account variables. First, we generalise the idea of account owner to that of *Comptroller*. For lender accounts, *Comptroller* and *Owner* are always the same. For borrowers however, taking a loan means ceding *Comptroller* role to the lender, and with it the ability to dispose of account's contents.

(5) $$Comptroller_B = \begin{cases} Owner_B & if\ BorrowBalance_{V,L,B} = 0\ \forall\, V, L \\ Owner_L & otherwise \end{cases}$$

(6) $$Comptroller_L = Owner_L$$

We further define *ActiveTerms* of an account as *Terms* that are permitted to change its state at a given time. As described above, *Terms* effectively contain owner's delegated authority. Thus, transfer of direct authority from one address to another under the comptroller mechanism is mirrored by a corresponding transfer of delegated authority.

(7) $$ActiveTerms_B = \begin{cases} Terms_B & if\ BorrowBalance_{V,L,B} = 0\ \forall\, V, L \\ Terms_L & otherwise \end{cases}$$

(8) $$ActiveTerms_L = Terms_B$$

Finally, we have the following definition of *AuthorityCheck*, where *Executor* is the EOA or smart contract address that executes the batch.

(9) $$AuthorityCheck = \begin{cases} Pass\ \ if\ Executor = Comptroller\ \vee\ Msg.Sender = ActiveTerms \\ Fail \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad otherwise \end{cases}$$

2.6 State Transition

Protocol state can only be modified through methods outlined below, executed within a batch. Economic security of the protocol relies primarily on two principles:

- Full backing of all synthetic assets issued by vaults with corresponding tokens
- Integrity and uniqueness of authority over individual accounts, under the definition provided above

To enhance user experience and without prejudice to security, the first principle is enforced at the end of every batch, effectively enabling native flash loans over all tokens contained in vaults. The same does not apply on individual account level – by default, account state variables cannot be modified at any point within a batch by anyone lacking authority. For context, owners could trivially enable flash loans for synthetic assets in their account by adding appropriate logic to account's *Terms*.

In contrast to a regular blockchain address, full control over base layer account state also implies that, by default, a third party cannot send assets to an account without its owner's permission. Again, this functionality can be trivially enabled through the account's *Terms*.

Methods below are described in terms of access control (who can execute them), state transitions they affect, and checks to be carried out immediately after state transitions and at the end of batch (in its verification phase).

There is only one context where a check is carried out immediately after state transition – in method (14) – to enforce uniqueness of authority. Checks at the end of batch play two roles – to enforce full backing of synthetic assets, and to ensure that entering a borrower-lender relationship is done with the consent of both parties.

Please note that methods listed below do not include supporting methods that carry no economic significance, but enable composability with external protocols.

2.6.1 Vault Methods

For a given vault V and $\Delta > 0$:

(10) $Deposit$

$Access\ control$: na
$State\ transition$: $TokenBalance_V = TokenBalance_V + \Delta$
$After\ method$: na
$End\ of\ batch$: na

(11) $Withdraw$

$Access\ control$: na
$State\ transition$: $TokenBalance_V = TokenBalance_V - \Delta$
$After\ method$: na
$End\ of\ batch$: $TokenBalance_V \geq TotalSyntheticBalance_V$

2.6.2 SyntheticBalance Methods

For a given borrower or lender account, and $\Delta > 0$:

(12) $mintSyntheticBalance$

$Access\ control$: $AuthorityCheck$
$State\ transition$ 1: $SyntheticBalance_V = SyntheticBalance_V + \Delta$
$State\ transition$ 2: $TotalSyntheticBalance_V = TotalAssetBalance_V + \Delta$
$After\ method$: na
$End\ of\ batch$: $TokenBalance_V \geq TotalSyntheticBalance_V$

(13) $burnSyntheticBalance$

$Access\ control$: $AuthorityCheck$
$State\ transition$ 1: $SyntheticBalance_V = SyntheticBalance_V - \Delta$
$State\ transition$ 2: $TotalSyntheticBalance_V = TotalSyntheticBalance_V - \Delta$
$After\ method$: na
$End\ of\ batch$: na

2.6.3 BorrowBalance Methods

For a given borrower account and $\Delta > 0$:

(14) $mintBorrowBalance$

$Access\ control$: $AuthorityCheck$
$State\ transition$: $BorrowBalance_{V,L,B} = BorrowBalance_{V,L,B} + \Delta$
$After\ method$: $BorrowBalance_{V,L',B} = 0\ \forall L' \neq L$
$End\ of\ batch$: $BorrowBalance_{V,L,B} = LendBalance_{V,L,B}$

(15) $burnBorrowBalance$

$Access\ control$: $AuthorityCheck$
$State\ transition$: $BorrowBalance_{V,L,B} = BorrowBalance_{V,L,B} - \Delta$
$After\ method$: na
$End\ of\ batch$: $BorrowBalance_{V,L,B} = LendBalance_{V,L,B}$

2.6.4 LendBalance Methods

For a given lender account and $\Delta > 0$:

(16) $mintLendBalance$

$Access\ control$: $AuthorityCheck$
$State\ transition$: $LendBalance_{V,L,B} = LendBalance_{V,L,B} + \Delta$
$After\ method$: na
$End\ of\ batch$: $BorrowBalance_{V,L,B} = LendBalance_{V,L,B}$

(17) $burnLendBalance$

$Access\ control$: $AuthorityCheck$
$State\ transition$: $LendBalance_{V,L,B} = LendBalance_{V,L,B} - \Delta$
$After\ method$: na
$End\ of\ batch$: $BorrowBalance_{V,L,B} = LendBalance_{V,L,B}$

3 PRODUCT-LEVEL INTUITION

3.1 Introduction

The above methods are intended as building blocks for more complex logic, spanning a vast product design space. Below, we provide a few examples of how to combine them into more practical operations and incorporate them into simple lending product functionality.

3.2 Moving Tokens between EOA and Lending Base Layer

Moving tokens into a base layer account involves two operations grouped into a single batch – depositing tokens into their vault (assuming the vault already exists) and minting a corresponding amount of synthetic assets in one's account. Atomicity is critical in this context, since depositing tokens into the vault without immediately claiming them means leaving them on the table for anyone to take. Moving tokens out of the base layer and between base layer accounts rely on similar logic:

- **EOA to Account**: [**execution**: $Deposit, mintSyntheticBalance$ | **verification**: $TokenBalance_V \geq TotalSyntheticBalance_V$]

- **Account to EOA**: [**execution**: $burnSyntheticBalance, Withdraw$ | **verification**: $TokenBalance_V \geq TotalSyntheticBalance_V$]
- **Account to Account**: [**execution**: $burnSyntheticBalance, mintSyntheticBalance$ | **verification**: $TokenBalance_V \geq TotalSyntheticBalance_V$]

3.3 Borrowing

While operations defined in the previous section are applicable across most potential use cases, borrowing logic can be constructed in a variety of ways. Below, we outline one simple example, devoid of much of the typical context, where a lender is willing to extend credit provided that borrowers' loan-to-value ratio does not exceed some *maxLTV* parameter.

We construct a *Borrow* method as a sequence of native base layer methods, to be included in lender account *Terms*, whereby the lender permits borrowers to burn synthetic assets in its account in return for minting of corresponding debt variables. For the sake of simplicity, we omit the mechanism for tracking relative value of borrow/lend balances vs the synthetic asset, and thus sizing the mint & burn.

We specify no access control other than the native *AuthorityCheck*, thus allowing arbitrary borrowers to interact with the method. Assuming the sample batch is signed by borrower account owner and no prior loans, *AuthorityCheck* is passed through *Executor=Comptroller* for *mintBorrowBalance* and *mintSyntheticBalance(B)*, and through *Msg.Sender=ActiveTerms* for remaining base layer methods.

In the sample batch, we omit checks performed as part of the native base layer methods and only highlight the one specified by our custom method.

- **Borrow**: *mintBorrowBalance→ mintLendBalance → burnSyntheticBalance(L) → requestCheck*
- **Sample batch:** [**execution**: *Borrow, mintSyntheticBalance(B)* | **verification**: *LTV ≤ maxLTV*]

3.4 Liquidity Provision and Rehypothecation

A simple mechanism that enables third parties to supply liquidity to a lender account could comprise a method under lender's *Terms* whereby users increment lender's synthetic balance in a particular asset and, in return, receive newly minted liquidity provider (LP) tokens from an external contract controlled by the lender account owner. Accepting these LP tokens as collateral under the *Borrow* method above would constitute rehypothecation.

We define *V* and *V`* as vaults for the token in which liquidity is denominated and the corresponding LP token, respectively, and use the same indices for the assets in these vaults. We define *mintLPTokens* as minting of new LP tokens by lender's owner, and the subsequent *Deposit* as moving these LP tokens into the vault.

- **addLiquidity(V)**: *mintSyntheticBalance(L, V) → mintLPTokens → Deposit(V`)*
- **Sample batch:** [**execution**: *addLiquidity, mintSyntheticBalance(B, V`)* | **verification**: *na*]

3.5 Achieving Target Leverage

In this example, we combine methods outlined above a swap at an external AMM to achieve a borrower-defined leverage target. This outcome normally requires iterative borrowing, but can be achieved in a single step due to the fact that the LTV check is only performed at the end of the batch.

We assume a borrower account with some amount of ETH, and a goal increasing ETH exposure through USDC leverage.

- **Sample batch:** [**execution**: *Borrow(USDC), Withdraw(USDC), Swap(USDC→ETH), Deposit(ETH), mintSyntheticBalance(B,ETH)* | **verification**: $LTV \leq maxLTV$]

4 GOVERNANCE

The protocol is designed to provide a wide spectrum of governance options, both at the level of vaults and of accounts. At vault level, it offers the choice between a fully decentralised configuration and one with a controller. In the controlled version, creators can make use of standardised functionality for pausing/unpausing of vaults and for emergency withdrawal of assets. Alternatively, custom logic can also be specified. Controller functionality can subsequently be switched off, turning a controlled vault into a fully decentralised one.

Accounts have no built-in governance functionality. Instead, it is assumed that a much simpler and more flexible approach is to implement exogenous governance directly over the account owner address. In practice, this allows one to re-use existing DeFi governance protocols with minimal or no modifications, or to implement full decentralisation.

Lending Base Layer contains no native fee mechanism. The sole privilege reserved by its creators is control over a whitelist of blockchain addresses allowed to create lender accounts. Creation of borrower accounts, vaults and vault builders is permissionless.

5 REFERENCES

[1] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, Dan Robinson. Uniswap v3 Core. 2021.

[2] Mathis Gontier Delaunay, Paul Frambot, Quentin Garchery, Mattieu Lesbre. Morpho Blue Whitepaper. 2023.

[3] Hayden Adams, Moody Salem, Noah Zinsmeister, Sara Reynolds, Austin Adams, Will Pote, Mark Toda, Alice Henshaw, Emily Williams, Dan Robinson. Uniswap v4 Core [Draft]. 2023.

[4] Emilio Frangella, Lasse Herskind. Aave v3 Technical Paper, 2022.

[5] Hayden Adams, Noah Zinsmeister, Mark Toda, Emily Williams, Xin Wan, Matteo Leibowitz, Will Pote, Allen Lin, Eric Zhong, Zhiyuan Yang, Riley Campbell, Alex Karys, Dan Robinson. UniswapX. 2023.

[6] Euler Team. Euler Vault Kit Whitepaper. 2024

[7] Euler Team. Ethereum Vault Connector Whitepaper. 2024

6 DISCLAIMER

This paper is for general information purposes only. It does not constitute investment advice or a recommendation or solicitation to buy or sell any investment and should not be used in the evaluation of the merits of making any investment decision. It should not be relied upon for accounting, legal or tax advice or investment recommendations. Opinions reflected in this paper are subject to change without being updated.