Dual Pools Audit

Testnet Deploy Instructions

Deploy and Configure Comptroller

1. Deploy XDP token
    - File name: XVS.sol
    - Inputs: 20000000000000000000 (as example, doesn't really matter)
    - Instructions: Paste addr in comptroller.sol function getXDPAddress() ~line 1520
2. Deploy Chain Link Oracle
    - File name: VenusChainlinkOracle.sol
    - Instructions: setFeed() for dTokens (dBNB, dBTCB, dBUSD to start)
        - "dBNB","0x2514895c72f50D8bd4B4F9b1110F0D6bD2c97526"
        - "BTCB","0x5741306c21795FdCBb9b265Ea0255F499DFe515C
        - "BUSD","0x9331b55D9830EF609A2aBCfAc0FBCE050A52fdEa"
3. Deploy ComptrollerLens
    - File name: ComptrollerLens.sol
4. Deploy Comptroller
    - File name: Comptroller.sol
5. Deploy Unitroller
    - File name: Unitroller.sol
6. Configure Comptroller
    - _setPendingImplementation(Comptroller address) in Unitroller
    - _become(Unitroller address) in Comptroller
    - Using Unitroller address but Comptroller ABI:
        - setPriceOracle(VenusChainlinkOracle address)
        - setComptrollerLens(ComptrollerLens address)
        - May also add the following (optional):
            - _setPauseGaurdian: testers adders
            - closeFactor: 1000000000000000000
            - liquidationIncentive: 1100000000000000000
                - This gives 10% (excess of 100%) to liquidator
        - May also send XDP to Comptroller (to test XDP distributions later)

Deploy and Configure dTokens

1. Deploy Trade Model
    - File name: TradeModel.sol

    - Instructions: paste in VTokenInterfaces.sol/VTokenStorage for variable tradeModel ~ line 160
        - Also able to update this address in dToken _setTradeModel(new addr)
2. Deploy JumpRateModel
    - File name: JumpRateModel.sol
    - Inputs: 10000000000000000,150000000000000000,3000000000000000000,400000000000000000
        - (base: 1%, baseMultiplier: 15%, jumpMultiplier: 300%, kink: 40%)
3. Deploy dBNB
    - File name: VBNB.sol
    - Inputs: Comptroller, JumpRateModel, 1000000000000000000,"Dual Pool BNB","dBNB",18, testersAddress
4. Deploy Delegate (implementation for dBep20 tokens)
    - File name: VBep20Delegate.sol
    - Notes: This is the implementation for dBep20 (ex, dBTCB, dBUSD, etc) tokens. For testing you may use the same implementation for each dBep20 token.
5. Deploy Delegator (repeat for each dBep20 token)
    - File name: VBep20Delegator.sol
    - Inputs: 0x8301F2213c0eeD49a7E28Ae4c3e91722919B8B47,Comptroller,JumpRateModel,1000000000000000000,"Dual Pool BUSD","dBUSD",18,testerAddress,Delegate,becomeImplementationData
    - Notes: I am not too sure what becomeImplementationData is, when looking at how it relates to VBep20Delegate.sol it looks like it's unused. I just inputted the zero address and it worked.
6. Configure Comptroller for dTokens
    - _supportMarket(dToken)
    - _setCollateralFactor(dToken,0.60e18)
        - 0.60e18 is reasonable, allows user to borrow 60% of dTokens supplied underlying value
    - _setXDPSpeed(dToken, 1e18)
        - This distributes 1 XDP per block to the entire dToken market

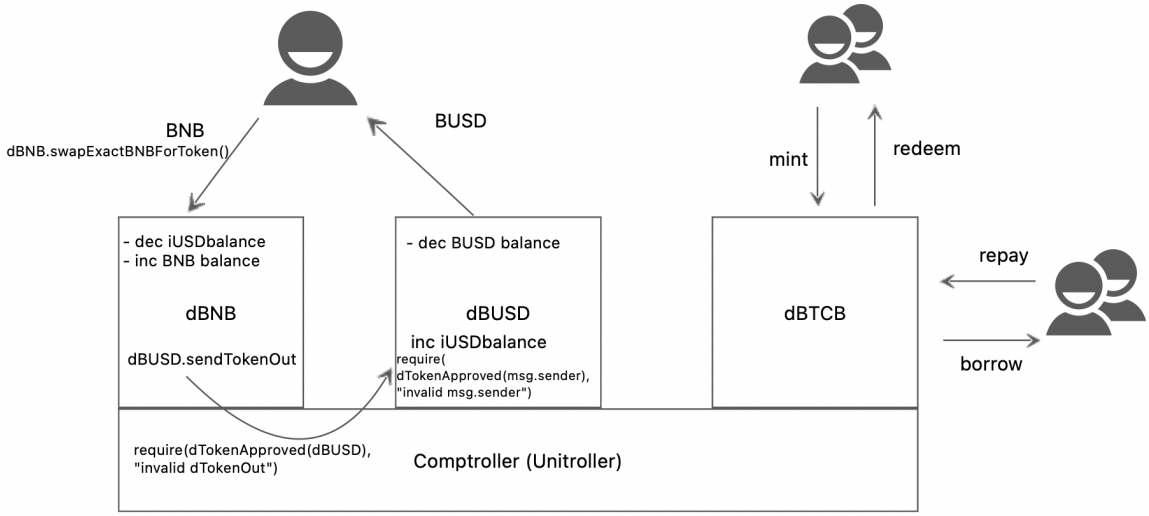
Potential Tests

- mint()/redeem()
    - Supply and redeem underlying from corresponding dToken market
- borrow()/repay()
    - Borrow and repay underlying from corresponding dToken
- liquidateBorrow()
    - Make sure able to liquidate borrow of underwater accounts and the liquidator receives liquidation incentive
- swapExactBNBForTokens()/swapExactTokensForTokens()
    - Tests selling BNB → token or token → token/BNB

- _puaseTrading()
    - Make sure it pauses buying or selling of this dToken
- _setProtocolPaused()
    - Make sure it pauses all actions of the protocol
- claimXDP()
    - Make sure XDP is distributed to dToken holders upon calling
- More functions! Thanks

Protocol Introduction

Dual Pools is a clone of the Venus Protocol. Dual Pools uses the supply/borrow functionality (lightly modified) of Venus, but also adds the ability to trade one currency for another. Traders can sell, for example, BNB to the dBNB pool (called vBNB on Venus) and receive the underlying asset from another pool such as BUSD from the dBUSD pool.



The modifications and additions are noted at the top of each file, with the primary modified files being

1. Comptroller.sol
2. VToken.sol
3. VBep20.sol/VBNB.sol
5. TradeModel.sol (addition)

I suggest comparing Dual Pools code to the Venus code (heavily audited and time tested) at and keep an eye on the modifications and additions.

Useful links:
https://docs.venus.io/docs/getstarted#introduction
https://github.com/VenusProtocol/venus-protocol/tree/master/contracts
https://dualpools.com/

XDP Token

The XDP token is an exact clone of the XVS token, except a name change, with no further modifications. It is the future governance token of the protocol

There is an additional use case for this token where users can gain a 10-70% trading fee discount based on how many XDP they hold in their wallet. This logic is found in the tradeModel file and does not require a modified XDP token contract.

File name: XVS.sol
XDP token address: 0x8549708b7c8dfAab00B5c5B97E483Caf008e4665

Price Oracle

Dual Pools uses the same Chainlink price oracle as Venus. There were no modifications or additions except removal of the VAI and XVS specific if-else statements in the getUnderlyingPrice() function

File name: VenusChainlinkOracle.sol
Oracle Address: 0x7966F821337Cdc8999c71a829d6676a93b7953c7

Comptroller Lens

There were no modifications or additions to the comptroller lens contract. This has additional code relating to the comptroller, but for contract size constraints Venus moved it to a separate contract.

File name: ComptrollerLens.sol
Comptroller Lens address: 0x4a2685A9F092d6DF0D6749327465460afa3Aad07

Comptroller

The comptroller is the risk management layers of the Venus Protocol which keeps track of the USD value of the supplied collateral users supplied across all dTokens. This sets the limit to how much the user may borrow (as determined by each dTokens collateral factors).

The comptroller also holds the code for the XDP (previously XVS) token distribution. The admin is able to set the venusSpeed (distribution rate) for each dToken which accrues XDP to each user who holds assets in a given dToken market.

The comptroller uses an awkward inheritance model. Following the deployment instructions should give you a good idea of the architecture.

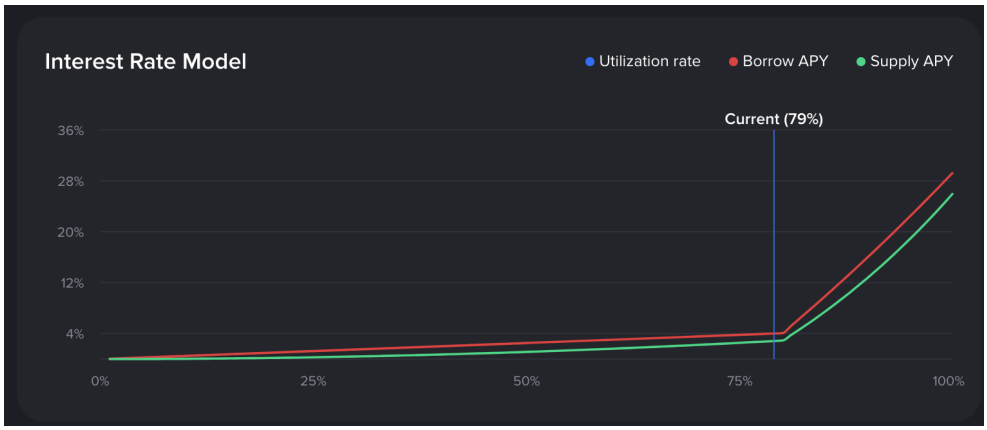
An addition was made to the comptroller to support trading functionality (bottom of contract). dTokens call the *dTokenApproved(address _dToken)* function in the comptroller which only returns true if the protocol is not paused, and the dToken is listed and not paused. This essentially allows dTokens to communicate and trust each other for trading purposes and trust.

File name: Comptroller.sol
Comptroller: 0x8B0608890632c191C23749683ae0E43dC0C8383A

JumpRateModel

The Jump Rate Model is the interest rate model Venus uses to determine the borrow and supply interest rate based on the utilization rate of the market (how much is borrowed relative to available assets in the pool). The JumpRateModel contract is upgradeable in the dToken contracts.

The JumpRateModel allows for a “kink” in the interest rate curve which allows a more rapid increase in borrow/supply rates after this specified utilization rate



File name: JumpRateModel.sol
JumpRateModel Address: 0x2AC8e741DDD6a0A9620d5ac7D1b6A616b9329916

TradeModel

The trade model is a complete addition to the Venus protocol to support trading functionality for Dual Pools. This is essentially the “brains” behind the trading model. Much like the interest rate model described above for lending/borrowing. The trade model contract is updatable in each dToken and only consists of view functions (no storage or contract changing functions).

The tradeModel holds functions related to:

i) swapping functionality - calculating how much of tokenB to send to user based on deposited tokenA
ii) redeem/borrow functionality - charging a fee when redeeming/borrowing based on how much liquidity was removed and how much this affected the iUSDrate
iii) trading fee discount - determine how much of a discount on trading fees to apply based on how much XDP the user holds

Swapping Functions

Rather than varying the supply/borrow rates according to a utilization rate like the interest rate model, the tradeModel varies the effective prices of underlying assets (discount or premium to the oracle price) based on the iUSDrate.

To understand the iUSDrate we must first understand the iUSDbalance (internal USD). The iUSDbalance is essentially an USD virtual balance within each dToken that stores the net cumulative buying/selling pressure of that dToken market. If the iUSD balance is negative then there has been a net cumulative selling pressure, vice versa for a positive iUSD balance.

iUSDrate = iUSDbalance / (cash*oraclePrice + iUSDbalance)

- Ranges from -100% to 100%

The price impact (relative to Oracle Price) is a direct function of the iUSDrate

Price impact: iUSDrate * abs(iUSDrate)

- There is also an option for the trade model admin to include a priceImpactLimit, for example of 80%.

Example 1 (negative iUSD balance):

Token: BUSD
iUSDbalance = -1000 ($1000 more of BUSD sold than purchased in the dBUSD market)
Cash = 11000 (total amount of BUSD in the market)
oraclePrice = $1.00

iUSDrate = -1000 / (11000*1.00 + -1000) = -10%
priceImpact = -10% * abs(-10%) = -1%

adjustedPrice = $1.00 * (1 - abs(0.01)) = 0.99

Due to the net selling pressure in that market as determined by the negative iUSDbalance, the price of BUSD is now $0.99 which should theoretically increase buying pressure

Example 2 (positive iUSD balance):

Token: BTCB
iUSDbalance = 1000 ($1000 more of BTCB purchased than sold in the dBTCB market)
Cash = 0.5 (total amount of BUSD in the market)
oraclePrice = $18 000

iUSDrate = 1000 / (0.50*18000 + 1000) = 10%
priceImpact = 10% * abs(10%) = 1%

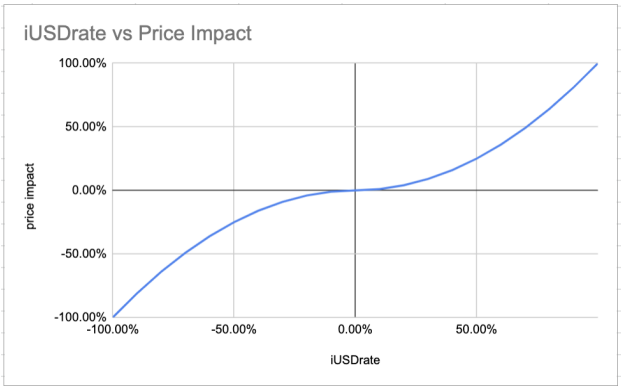
adjustedPrice = $18 000 / (1 - abs(0.01)) = 18 182

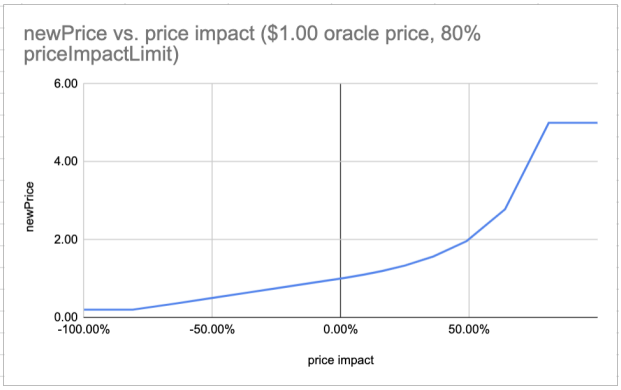
- Note, adjusted price formula varies * or / based on if the price impact is + or -

Due to the net buying pressure in that market as determined by the positive iUSDbalance, the price of BTCB is now ~$18 182 which should theoretically increase selling pressure.

The trade model and price impact formula may be updated and vary from token to token, for example the formula may differ between stablecoins and non-stablecoins.

Diagrams



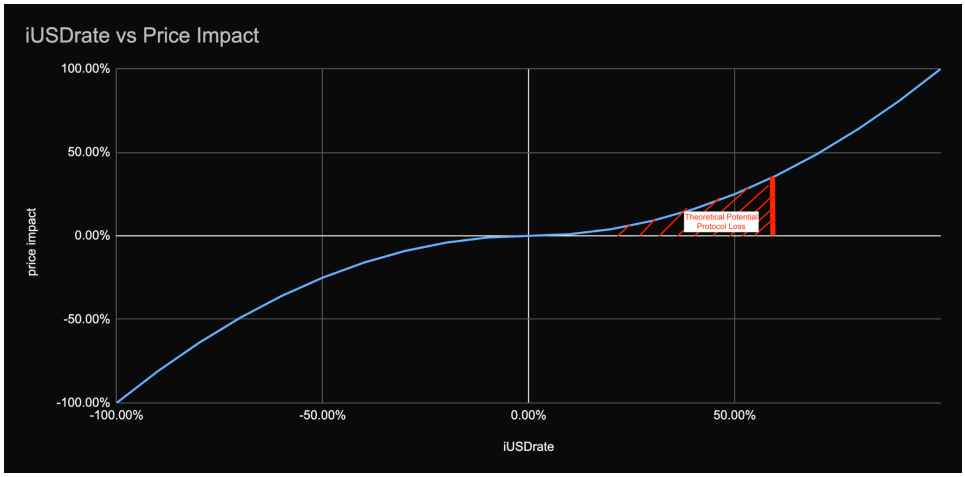


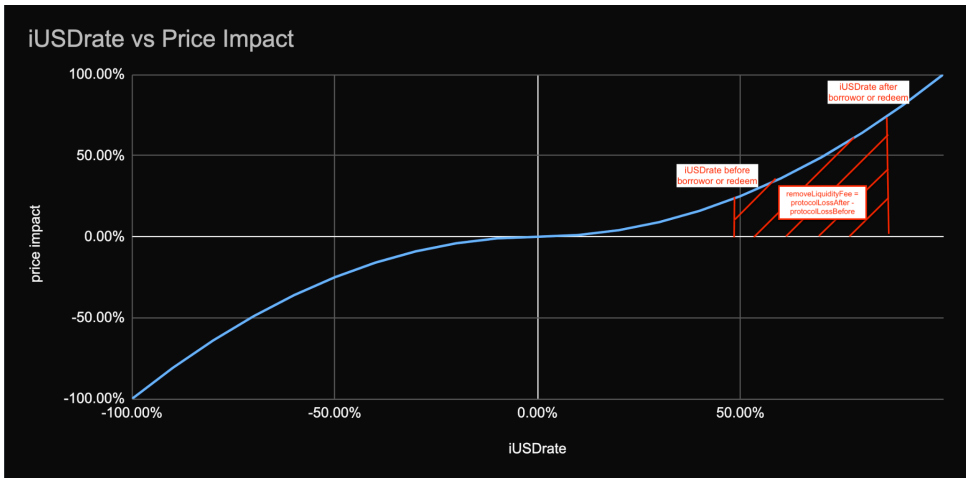
Ideally, the iUSDrate will stay between +/- 10% for each dToken. Arbitrage opportunities should help maintain iUSDrates within this range.

Redeeming and Borrowing Functions

In addition to functions relating to buying and selling underlying assets, the TradeModel also includes view functions to adjust the redeem and borrow (removing liquidity) from dToken contracts. This is to protect the exploit when the dToken has a nonzero iUSDrate and the user withdrawals liquidity from the pool (borrow or redeem) thereby resulting in a more extreme (more negative or more positive) iUSDrate. This would result in a greater price premium or discount that the user could take advantage of.

In order to protect against this exploit, the protocolLoss() value is calculated before and after the trade in removeLiquidityFee() and deducted from the users borrow or redeem to off-set this potential exploit. The protocolLoss() function is the theoretical amount the pool of assets could lose if a trader took advantage of the price discount or premium by making infinitesimal trades until the price discount or premium dissapeared.





Furthermore, the protocolLoss() function is used in cashAddUSDMinusLoss() which is ultimately used in the exchangeRate calculation in the dTokens. This effectively offsets the maximum theoretical trade loss due to the price premium/discount so the removeLiquidityFee is not realized until the iUSDrate approaches back to 0%. This behavior is more favorable as it may incentivize users to supply liquidity to dTokens with the most extreme iUSDrates which in return

decreases the iUSD rate as more liquidity is in the pool (cash liquidity is in denominator of iUSDrate formula)

Trading Fees

The trade model also holds functions relating to the trading fees and referral program. The referral program gives a referralDiscount (default 10%) discount to referrals. There is also a scaled XDP utility token discount from 10-70% based on how much XDP they hold in their wallet at the time of the trade.

tradeModel contract: 0xd835cB6545b5107F7A8273d81cc4cAA41266e95f

dTokens

The majority of the Venus Protocol logic as well as modifications/additions are in the dToken contracts. The dTokens allow users to mint (underlying → dTokens), redeem (dTokens → underlying), borrow, and repay the underlying asset. With Dual Pool additions, users can now sell an underlying asset to its corresponding dToken contract and receive the underlying asset of another dToken.

The primary modifications/additions to the dTokens are noted at the top of the VToken contract. The VBep20 and VBNB contracts inherent from the VToken contract and hold additional code to handle BEP20 or BNB specific functions. Some optional functions were removed to create enough room for trading functionality additions

dTokens also holds some logic regarding the referral program. The swapExactBNBForToken() and swapExactTokenForToken() functions send the reserveFee to the referrer. Although the dToken associated with the underlying token being purchased still sends the reserveFee to the reserves.

Primary Modifications and additions

1) Supply and borrow rates use available cash instead of total cash (getCashPrior()). This is so the rates increase as the absolute IUSDrate increases.
    a) availableCash = (getCashPrior() + iUSDbalance/getPriceToken) * (1e18 - abs(iUSDrate()))
2) The exchange rate formula uses exchange cash to include the maximum theoretical protocol loss. This is so the exchangeRate doesn't increase (much) as abs(iUSDrate) increases and decreases as abs(iUSDrate) decreases. Discussed in trade model.
    a) ExchangeCash = cash + (USD - protocolLoss)/oraclePrice
3) mintBehalfInternal(), mintBehalfFresh(), redeemInternal(), _addReservesInternal(), and _addReservesFresh() removed to make room for trading functionality. The

redeemTokensIn parameter in redeemFresh() and associated functionality was also removed due to removal of redeemInternal().

4) Integrations to the trade model were included at the bottom of the VToken contract. The minimum logic was included in these functions and instead relied on the trade model to decrease VToken contract size and improve upgradeability in the future as the tradeModel contract can easily be upgraded. Some safety required statements were put in place to ensure values from the trade model are reasonable in the event a malicious tradeModel address was updated.
5) Swap state-changing functions are included at the bottom of the VToken contract as well as the VBep20/VBNB contract.

VBep20 tokens have the option to have an upgradeable proxy model with the VBep20Delegate.sol and VBep20Delegator.sol contracts which is the option we used at Dual Pools. This is opposed to deploying VBep20 tokens with VBep20Immutable.sol. VBNB.sol is immutable, although the interestRateModel and TradeModel are upgradeable so there still remains some flexibility there.

File name: VToken.sol and VBep20Delegate.sol
vBep20Delegate (non-stable): 0x762b06eb432fb87df354801a3171f2bb6b8f9c08
vBEP20Delegate (stablecoin): 0x58a50E9AFFb6275716921bbfda6686996ef41b56

- Currently these two implementations are identical, although in the future stablecoin and non-stablecoin implementations may differ

File name: VBep20Delegator.sol
vBUSDDelegator: 0x1a276A66cf38F439aE8128a3735eC79e12E6ffe9
vBTCBDelegator: 0xBED992A075d24F6770F2e2Bacd523a484F768aC6

- dBUSD and dBTCB are identical except for the underlying token address

vBNB: 0x8Da8bEFb17086F34511359e226A2b0c9b93E6C36

Please note these verified contracts have been slightly altered to include the referral program. New dToken mainnet contracts will be updated after the audit.

If you have any questions feel to contact me via the following methods
Telegram: @JavisLockhart
Email: jay@tradeium.io