bPNM

by PHENOMENAL CLUB

SMART CONTRACT AUDIT

zokyo

March 28th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passed a security audit.

ZOKYO AUDIT SCORING BPNM

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:

- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:

- 0 Critical issues: 0 points deducted
- 0 High issues: 0 points deducted
- 1 Medium issue: 1 verified = 0 points deducted
- 1 Low issues: 1 verified = 0 points deducted
- 25 Informational issues: 11 resolved, 8 verified, 4 acknowledged = 2 points deducted
- 2 points are deducted for the fact that the marketplace administrator can always edit the price of the item

Thus, 100 - 2 - 2 = 96 points out of 100.

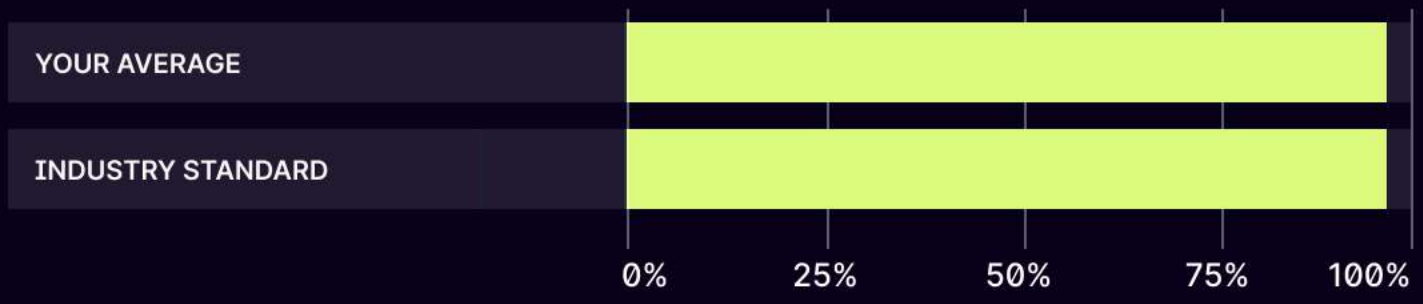
TECHNICAL SUMMARY

This document outlines the overall security of the bPNM smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the bPNM smart contract codebase for quality, security, and correctness.

Contract Status

Testable Code



Corresponds to the industry standards.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the BSC network's fast-paced and rapidly changing environment, we recommend that the bPNM team put in place a bug bounty program to encourage further active analysis of the smart contract.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the bPNM repository:
https://github.com/tguy6/bpnm_audit
Branch: main
Initial commit: ab499de741064e981c3623e25cfa3e3433310d5a
Final commit: 6457488834798a0c2e8b686625a9f33cc2b7beeb

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of bPNM smart contract. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contract logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contracts by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo Security team has conducted a detailed security audit of the bPNM project, a major component of the Phenomenal DeFi ecosystem. The bPNM protocol introduces the bPNM token. This deflationary utility token is minted upon purchase and burned upon sale and is backed by BTCB, a wrapped version of Bitcoin on the BNB chain. The initiation into the ecosystem is facilitated by acquiring a Limit Pack, which allows users to buy and sell bPNM tokens, engage with the marketplace, and participate in marketing programs.

Central to the bPNM ecosystem is the Phenomenal Tree, a multi-level marketing structure designed to allocate rewards and incentives based on user engagement and referrals. This ecosystem is further enriched by the inclusion of the GWT token, which unlocks additional functionalities as the extension of transaction limits and the reduction of fees. Another part of the protocol - Phenomenal Consultants NFT collection - offers users passive income opportunities and various discounts.

Throughout the audit, our team focused on assessing the integrity of smart contracts critical to the bPNM ecosystem's operations, including bpnmMain.sol, the cornerstone of the bPNM token system; PhenomenalConsultants.sol, which governs the NFT collection; and phenomenalLiquidityDistributor.sol (PLD), which manages liquidity distribution

Our findings highlighted several areas for improvement, including but not limited to the handling of payment tokens, user activation processes, and potential optimizations for gas efficiency and contract readability. The bPNM team resolved most of the issue. However, certain unresolved and acknowledged concerns require further action, particularly aligning technical implementation details with project documentation and whitepaper descriptions.

Executive Summary

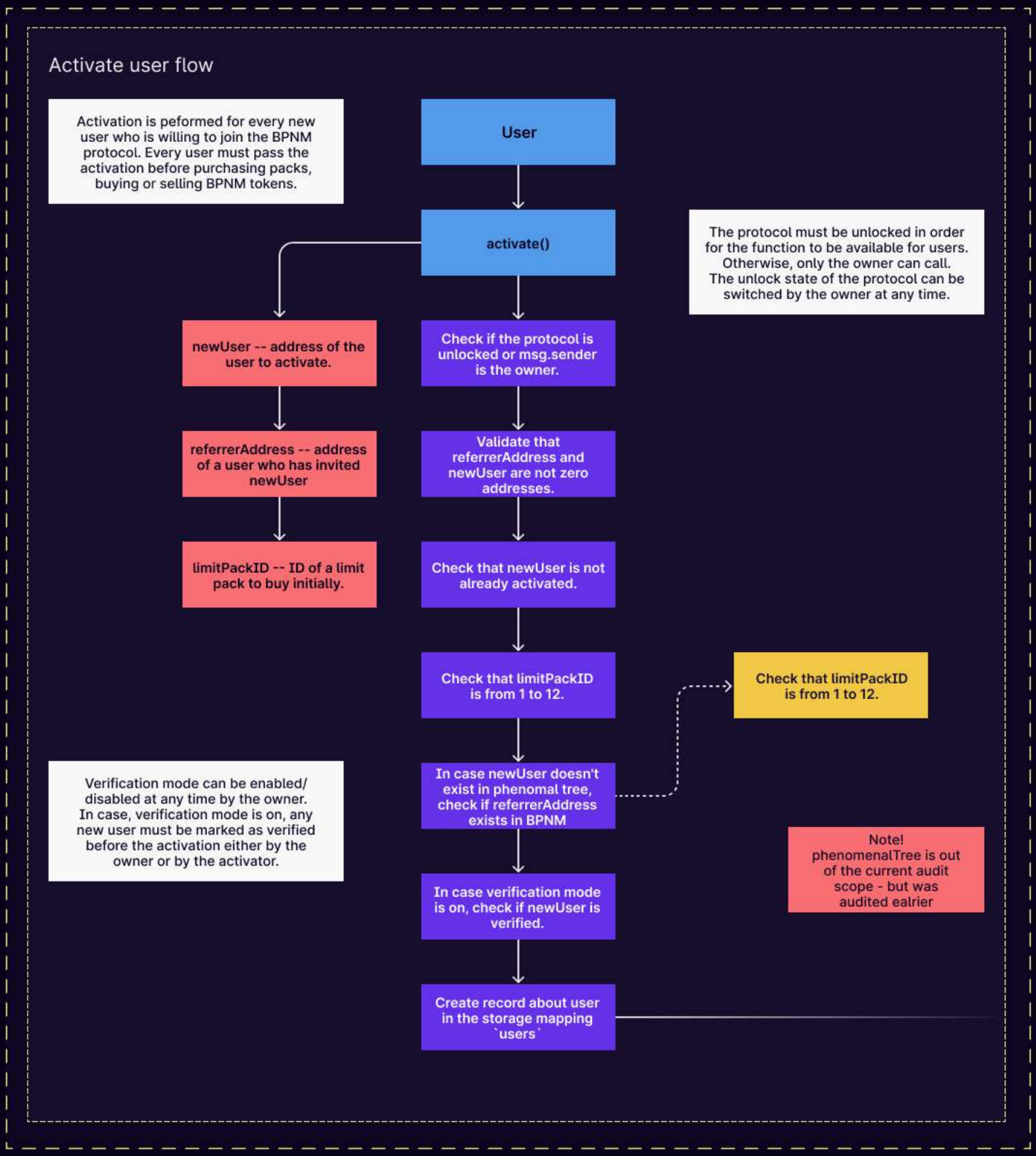
The audit confirmed the security of the protocol and validated its functionality to work as expected without disruptions and without significant risks.

Nevertheless, the audit resulted in a deduction of four points from the project's overall score due to unresolved issues:

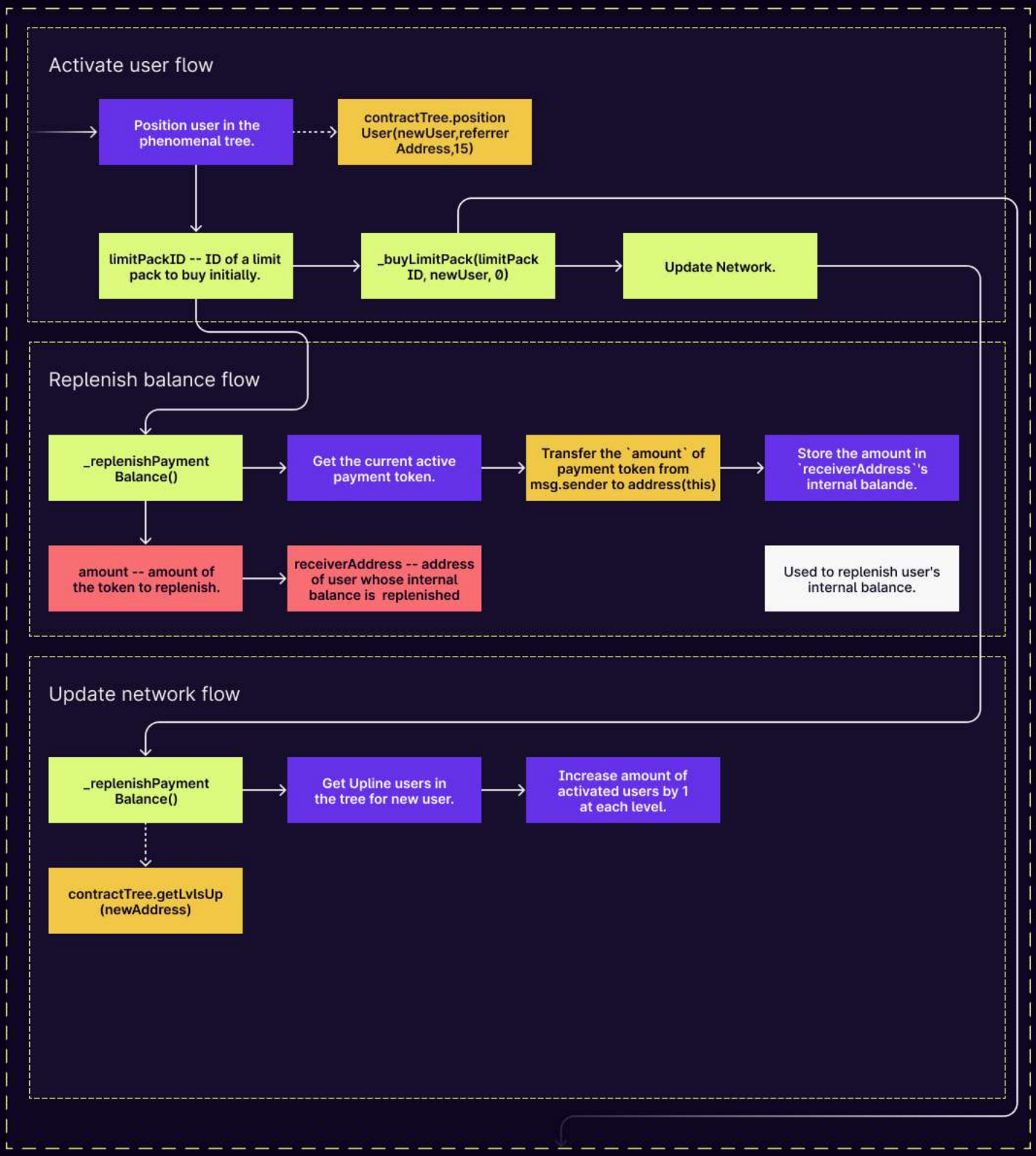
- Marketplace Administrator's Control (-2 point): The ability of the marketplace administrator to alter item prices at will introduces a risk of unfair pricing and manipulation in case of the admin key compromise.
- Inconsistent Naming Conventions (-1 points): The protocol violated Solidity style guide conventions and best practices, specifically in capitalizing Event and Struct names, undermines code readability potentially affecting long-term support and further development.

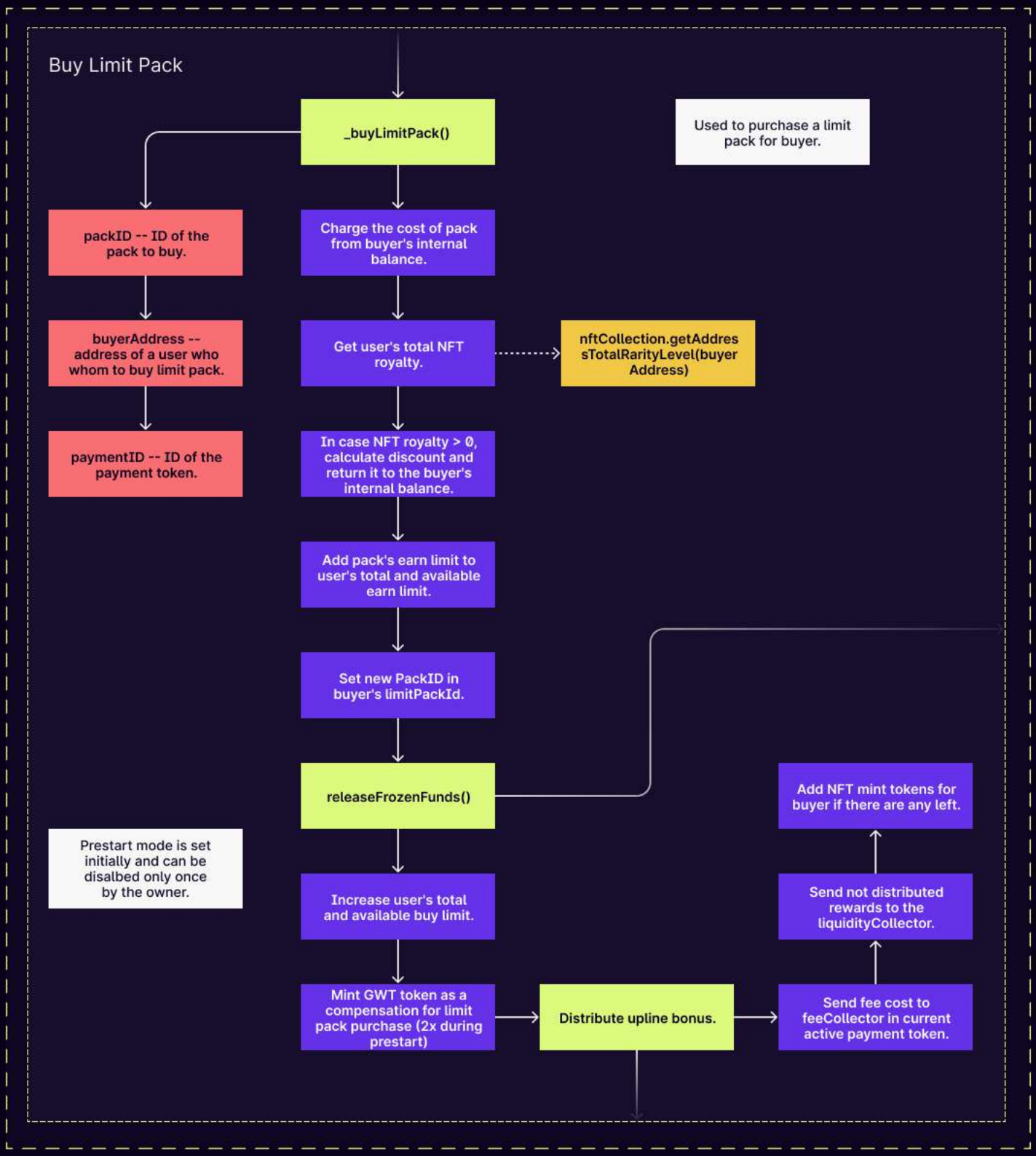
Also the security team leaves concerns regarding the testnet specification currently hardcoded in contracts, and unvalidated work with decimals - both issues were verified by the team, though still require attention during further development.

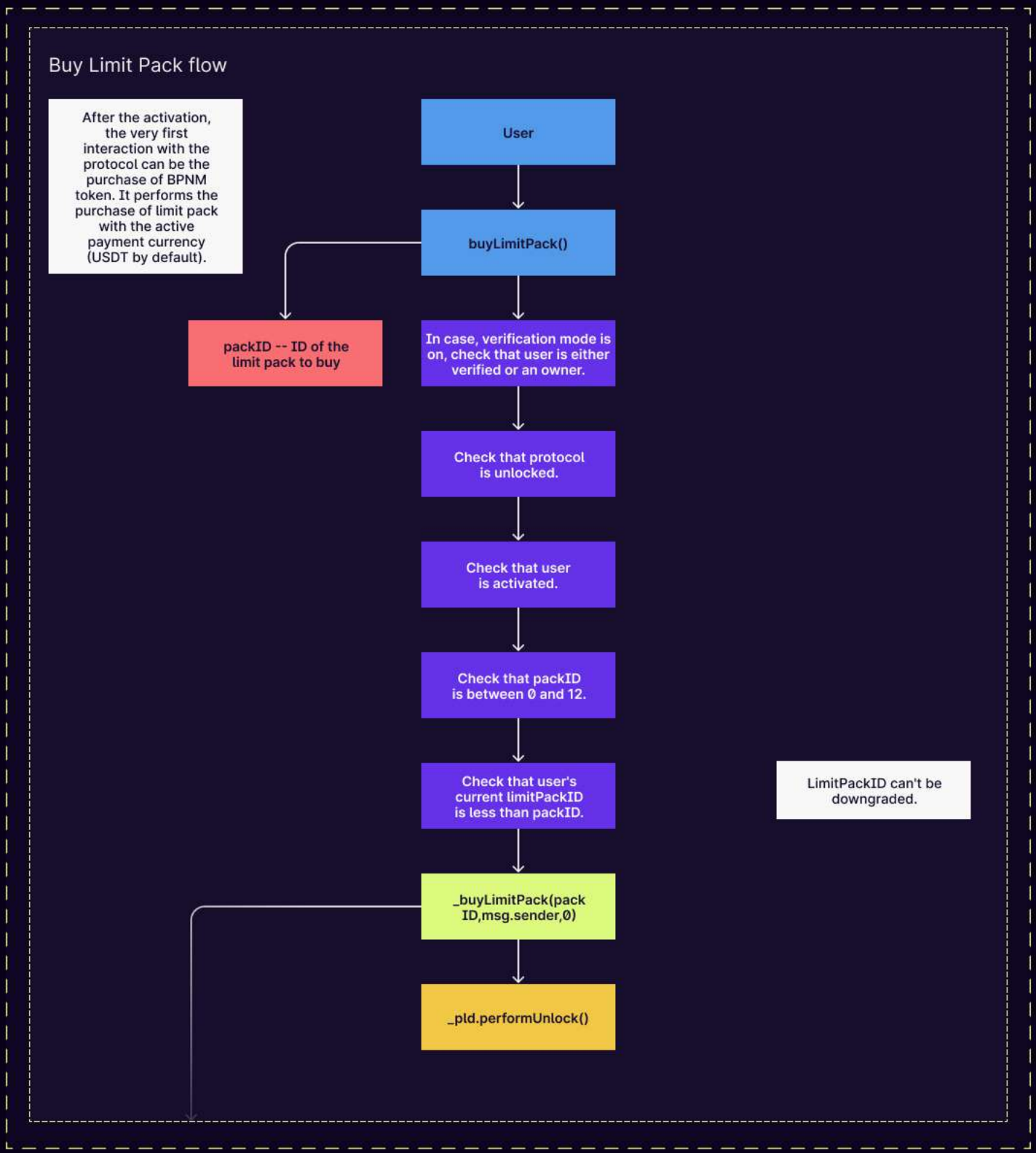
From other points of view contracts have high level of security and keep the integrity of performed functionality.

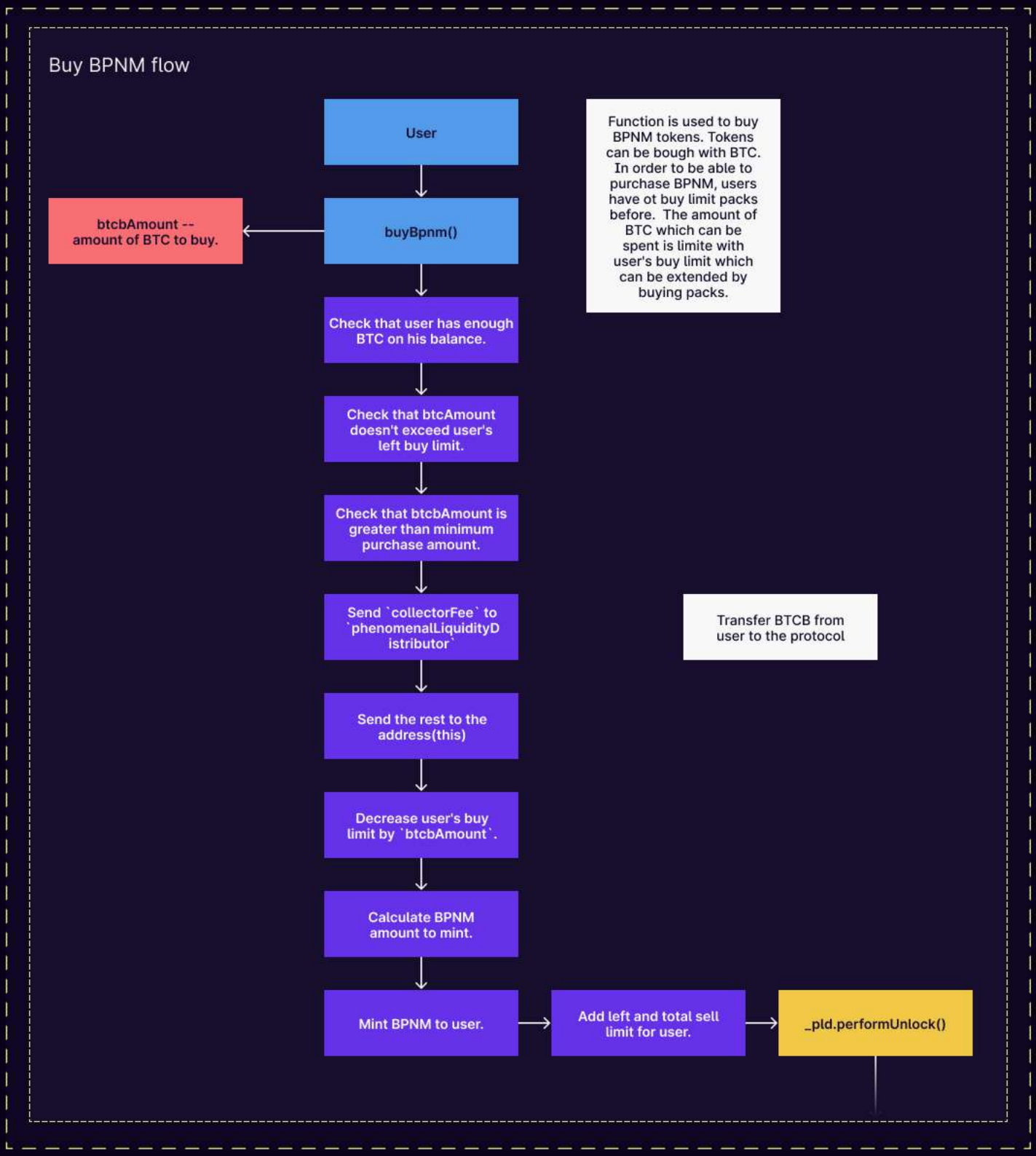

Activate user flow
Activation is peformed for every new user who is willing to join the BPNM protocol. Every user must pass the activation before purchasing packs, buying or selling BPNM tokens.
User
activate()
The protocol must be unlocked in order for the function to be available for users. Otherwise, only the owner can call. The unlock state of the protocol can be switched by the owner at any time.
newUser -- address of the user to activate.
referrerAddress -- address of a user who has invited newUser
limitPackID -- ID of a limit pack to buy initially.
Check if the protocol is unlocked or msg.sender is the owner.
Validate that referrerAddress and newUser are not zero addresses.
Check that newUser is not already activated.
Check that limitPackID is from 1 to 12.
Check that limitPackID is from 1 to 12.
In case newUser doesn't exist in phenomal tree, check if referrerAddress exists in BPNM
Verification mode can be enabled/ disabled at any time by the owner. In case, verification mode is on, any new user must be marked as verified before the activation either by the owner or by the activator.
Note! phenomenalTree is out of the current audit scope - but was audited ealrier
In case verification mode is on, check if newUser is verified.
Create record about user in the storage mapping `users`

bPNM.sol

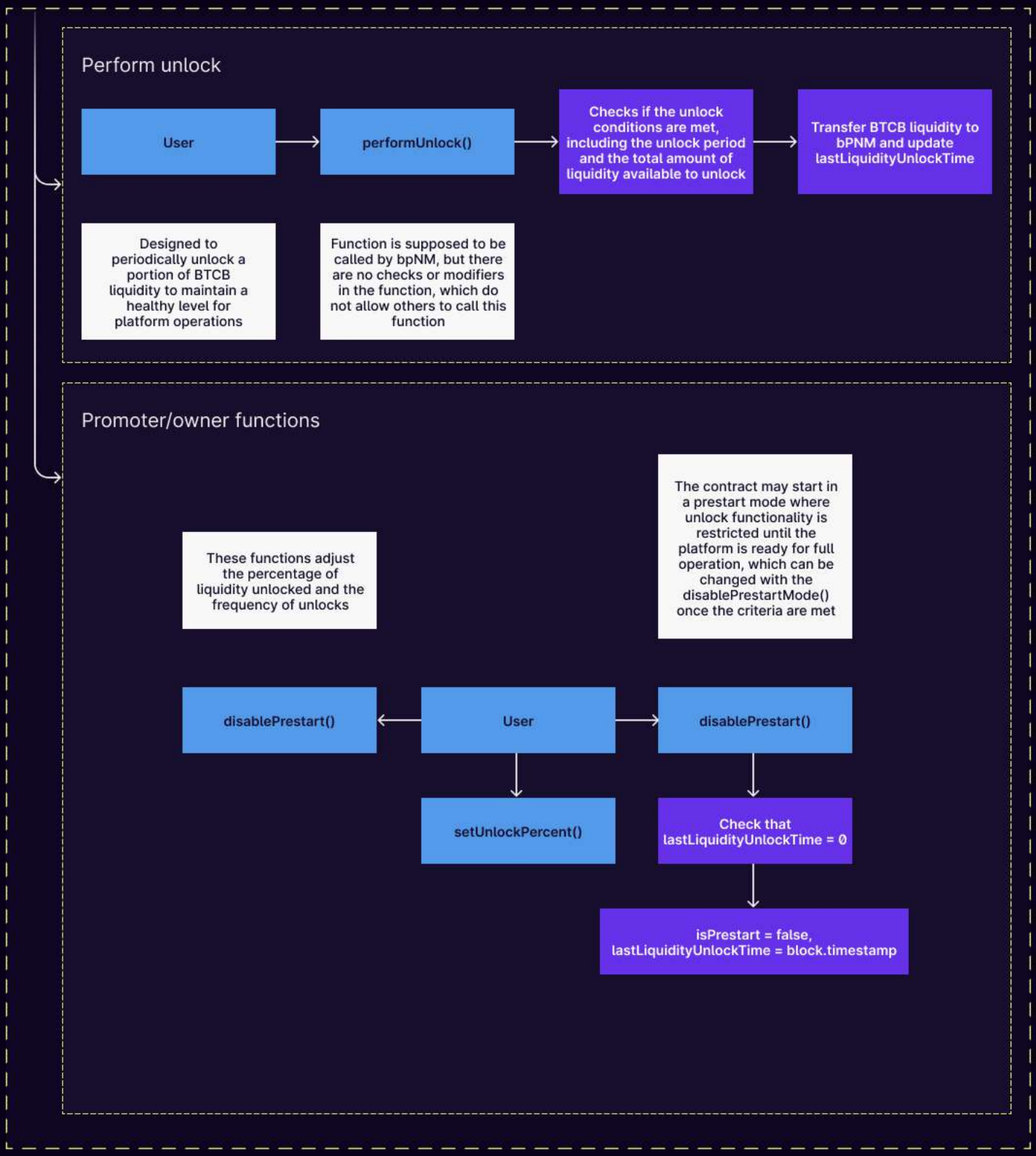

Activate user flow
Position user in the phenomenal tree.
contractTree.position User(newUser,referrer Address,15)
limitPackID -- ID of a limit pack to buy initially.
_buyLimitPack(limitPack ID, newUser, 0)
Update Network.
Replenish balance flow
_replenishPayment Balance()
Get the current active payment token.
Transfer the `amount` of payment token from msg.sender to address(this)
Store the amount in `receiverAddress`'s internal balande.
amount -- amount of the token to replenish.
receiverAddress -- address of user whose internal balance is replenished
Used to replenish user's internal balance.
Update network flow
_replenishPayment Balance()
Get Upline users in the tree for new user.
Increase amount of activated users by 1 at each level.
contractTree.getLvlsUp (newAddress)

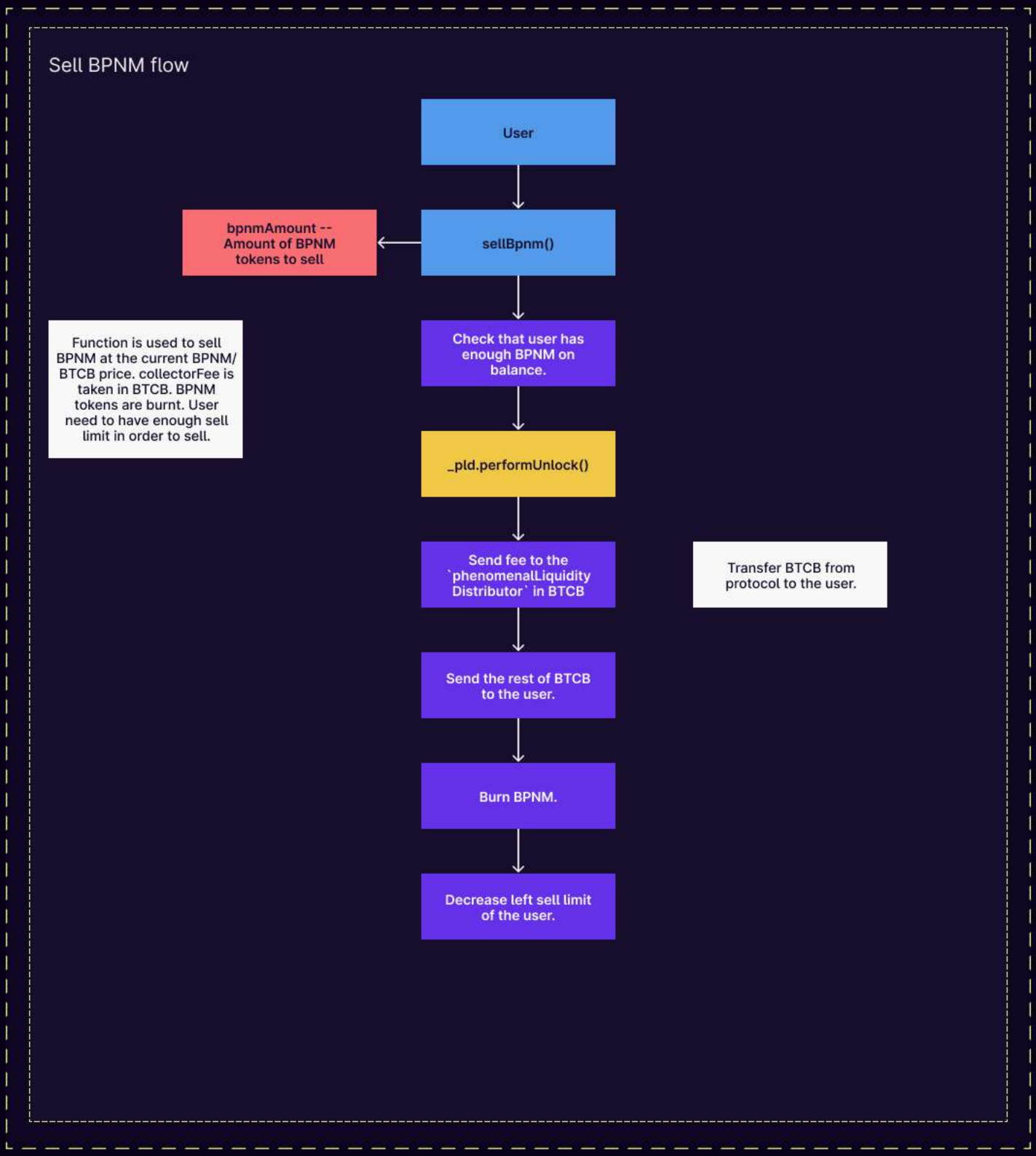

Buy Limit Pack
_buyLimitPack()
Used to purchase a limit pack for buyer.
packID -- ID of the pack to buy.
Charge the cost of pack from buyer's internal balance.
buyerAddress -- address of a user who whom to buy limit pack.
Get user's total NFT royalty.
nftCollection.getAddressTotalRarityLevel(buyerAddress)
paymentID -- ID of the payment token.
In case NFT royalty > 0, calculate discount and return it to the buyer's internal balance.
Add pack's earn limit to user's total and available earn limit.
Set new PackID in buyer's limitPackId.
releaseFrozenFunds()
Add NFT mint tokens for buyer if there are any left.
Prestart mode is set initially and can be disalbed only once by the owner.
Increase user's total and available buy limit.
Send not distributed rewards to the liquidityCollector.
Mint GWT token as a compensation for limit pack purchase (2x during prestart)
Distribute upline bonus.
Send fee cost to feeCollector in current active payment token.


Buy Limit Pack flow
After the activation, the very first interaction with the protocol can be the purchase of BPNM token. It performs the purchase of limit pack with the active payment currency (USDT by default).
User
buyLimitPack()
packID -- ID of the limit pack to buy
In case, verification mode is on, check that user is either verified or an owner.
Check that protocol is unlocked.
Check that user is activated.
Check that packID is between 0 and 12.
Check that user's current limitPackID is less than packID.
LimitPackID can't be downgraded.
_buyLimitPack(packID,msg.sender,0)
_pld.performUnlock()

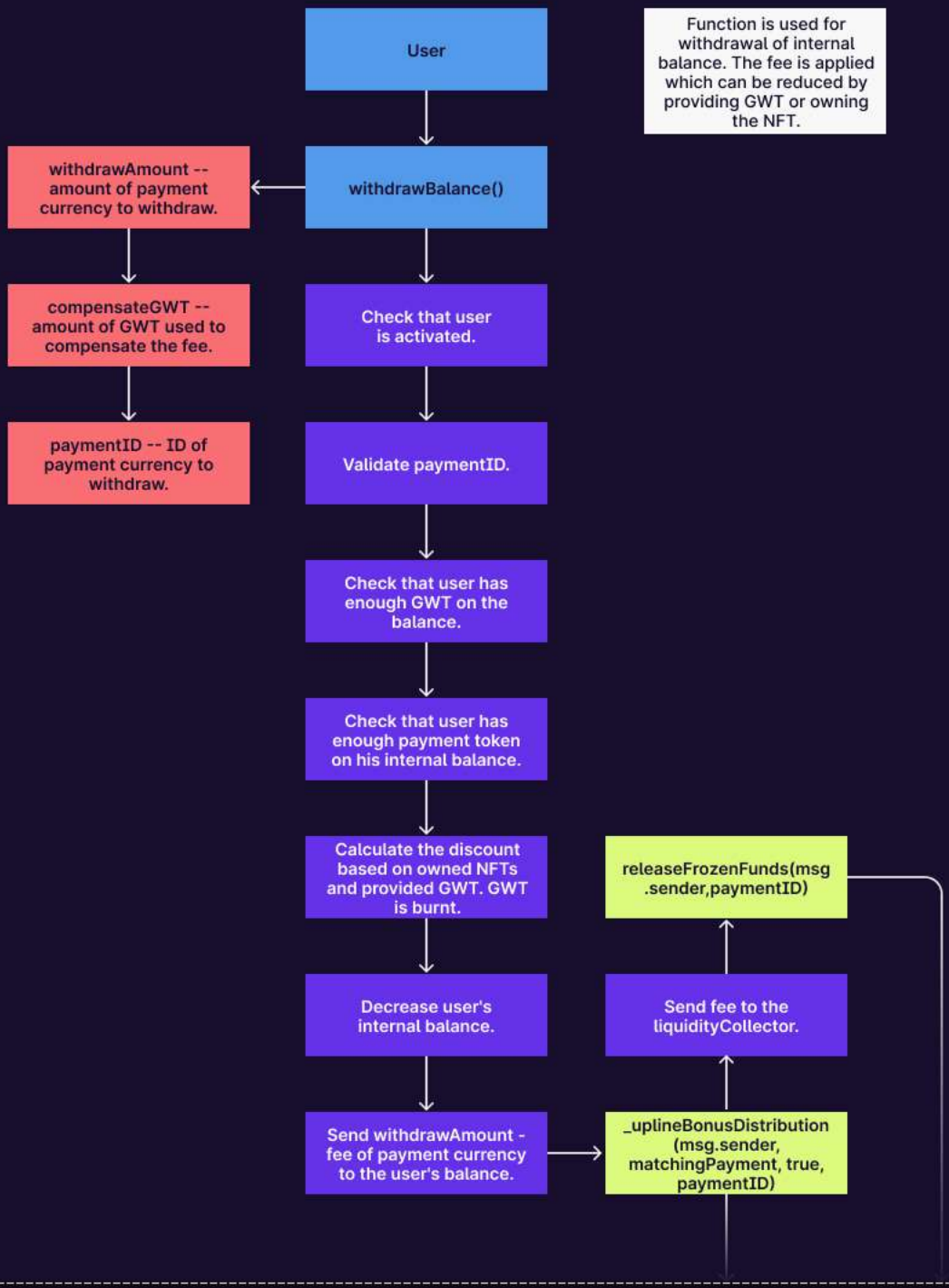

Buy BPNM flow
User
buyBpnm()
btcbAmount -- amount of BTC to buy.
Function is used to buy BPNM tokens. Tokens can be bough with BTC. In order to be able to purchase BPNM, users have ot buy limit packs before. The amount of BTC which can be spent is limite with user's buy limit which can be extended by buying packs.
Check that user has enough BTC on his balance.
Check that btcAmount doesn't exceed user's left buy limit.
Check that btcbAmount is greater than minimum purchase amount.
Send `collectorFee` to `phenomenalLiquidityDistributor`
Transfer BTCB from user to the protocol
Send the rest to the address(this)
Decrease user's buy limit by `btcbAmount`.
Calculate BPNM amount to mint.
Mint BPNM to user.
Add left and total sell limit for user.
_pld.performUnlock()

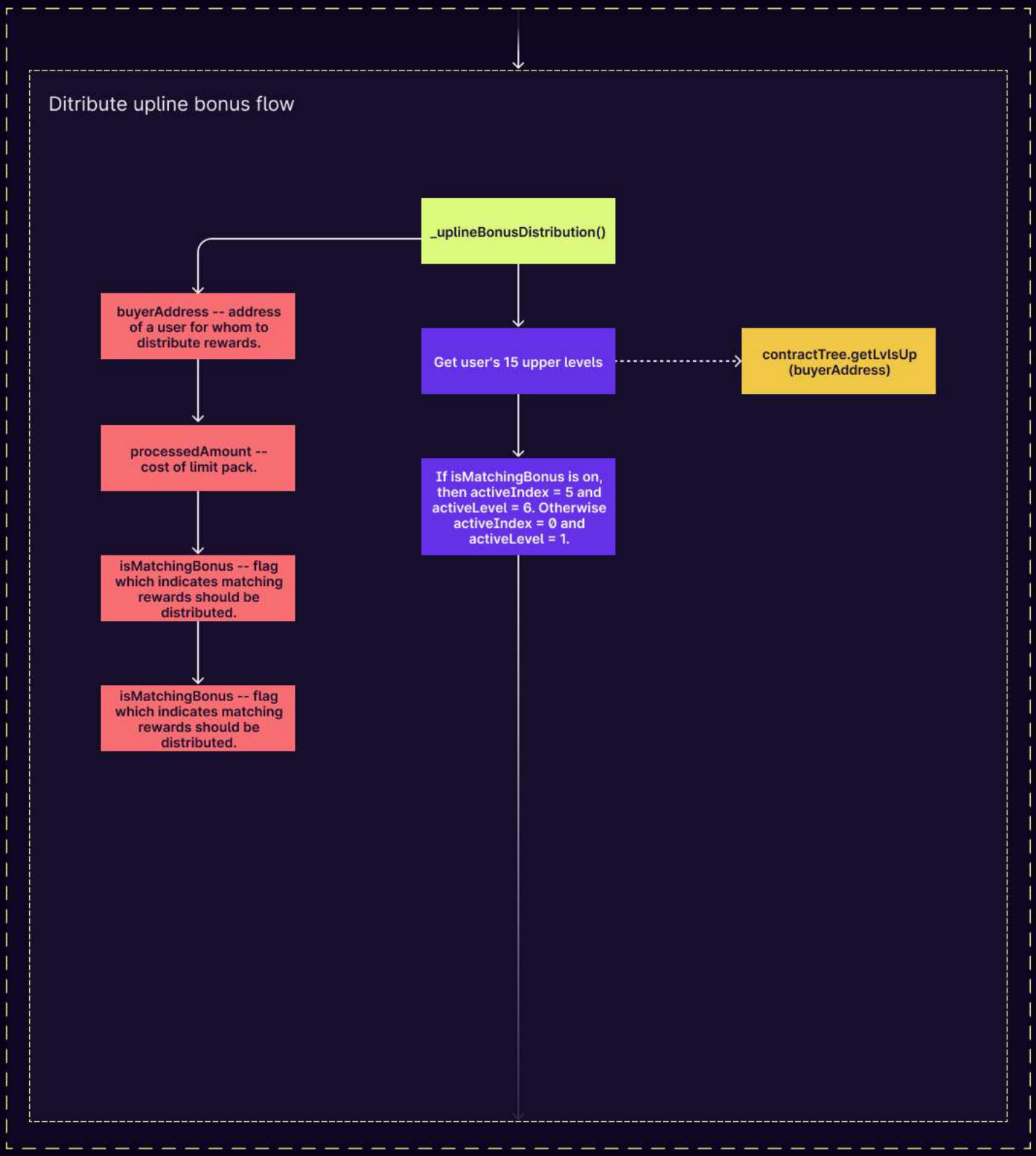
phenomenalLiquidityDistributor

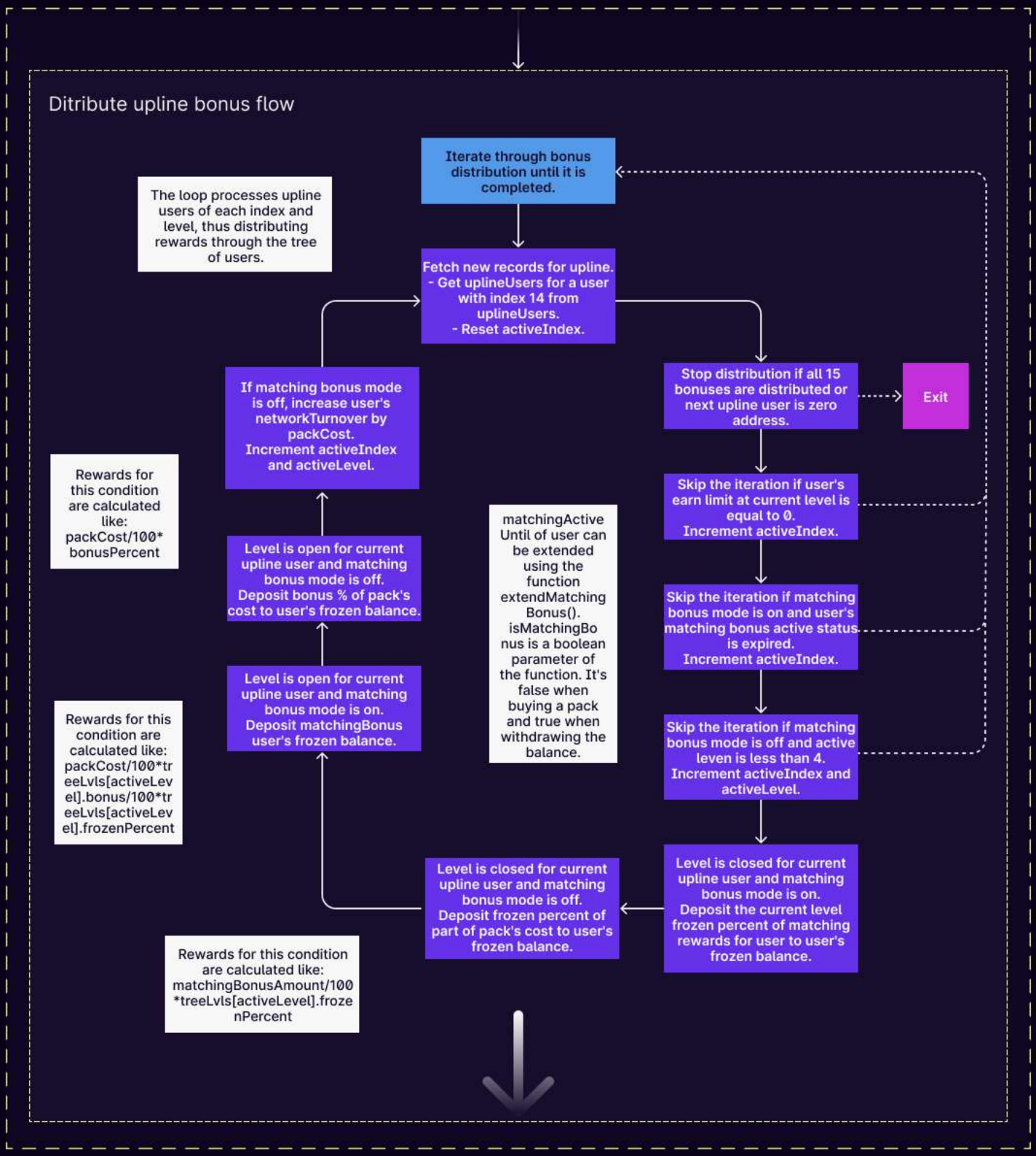


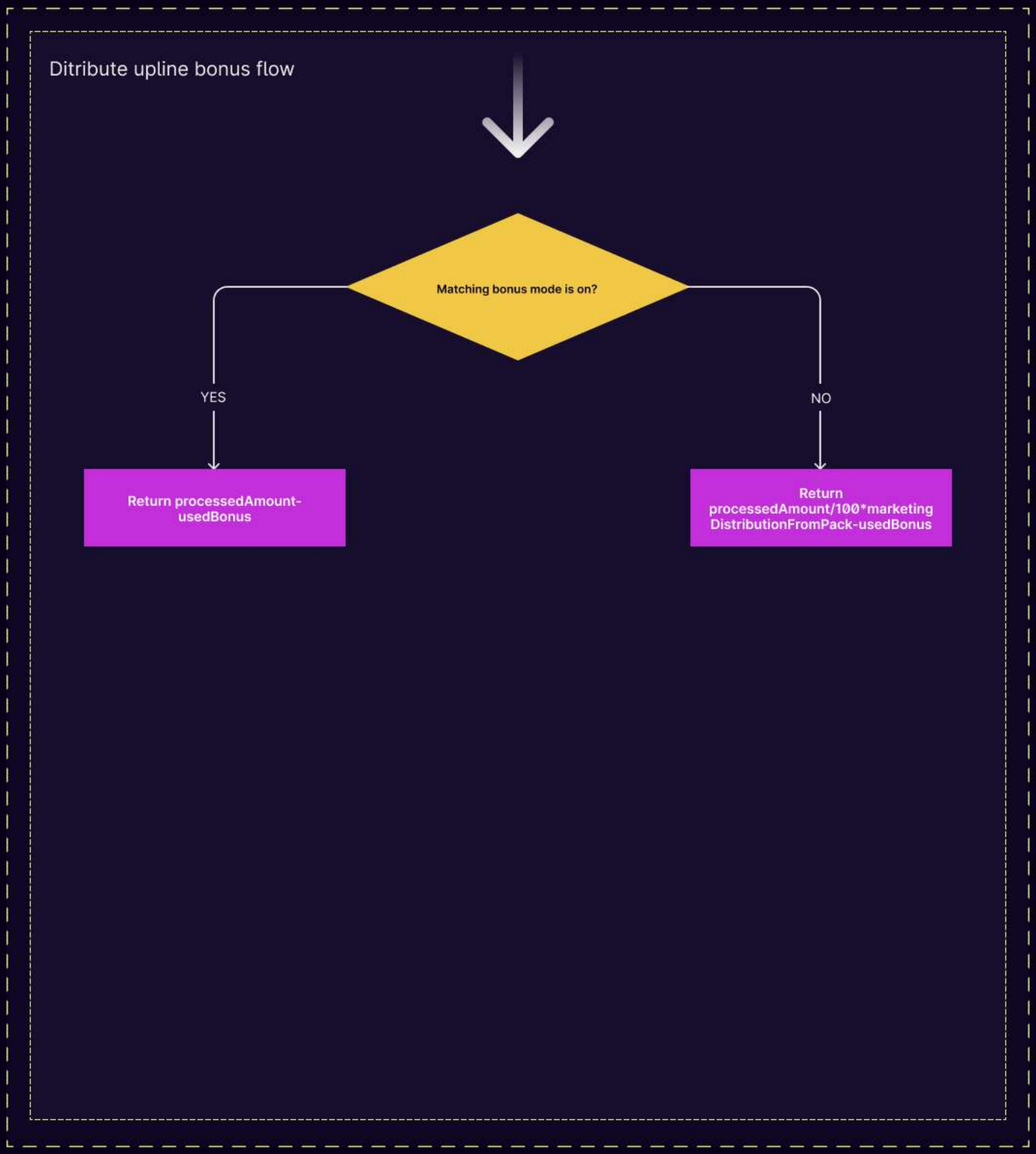

Sell BPNM flow
User
sellBpnm()
bpnmAmount -- Amount of BPNM tokens to sell
Function is used to sell BPNM at the current BPNM/ BTCB price. collectorFee is taken in BTCB. BPNM tokens are burnt. User need to have enough sell limit in order to sell.
Check that user has enough BPNM on balance.
_pld.performUnlock()
Send fee to the `phenomenalLiquidity Distributor` in BTCB
Transfer BTCB from protocol to the user.
Send the rest of BTCB to the user.
Burn BPNM.
Decrease left sell limit of the user.

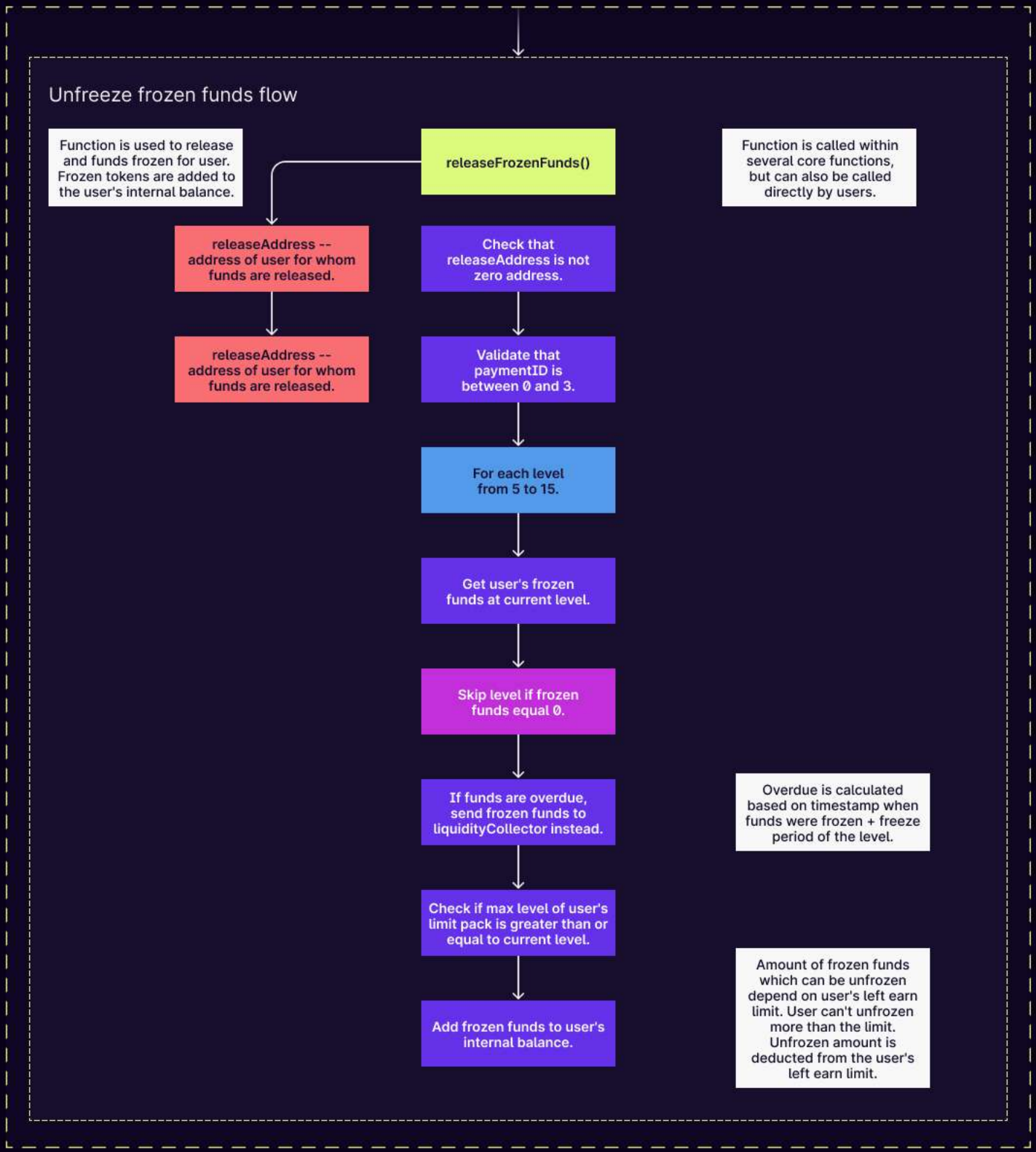
Withdraw internal balance flow

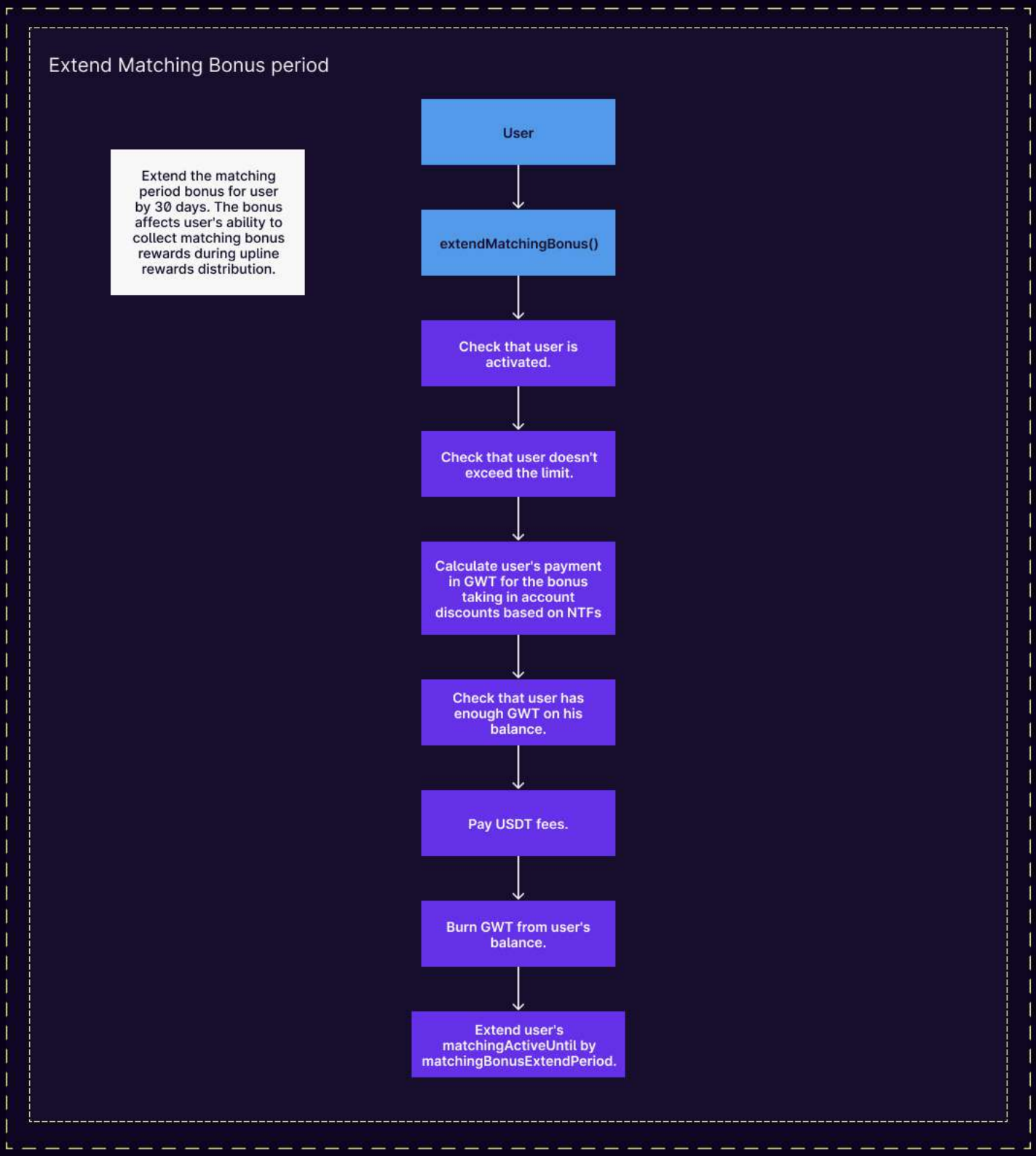


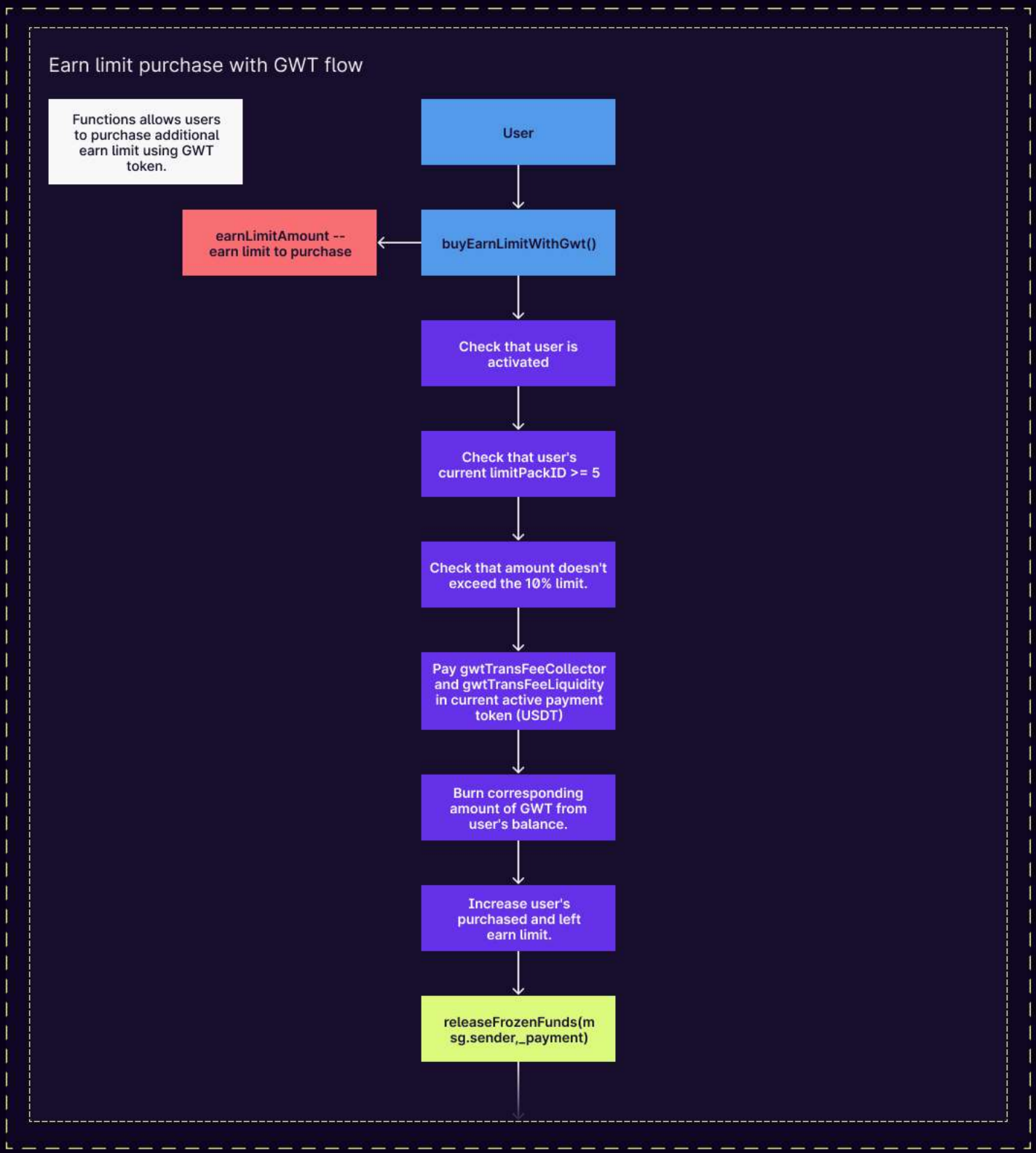

Ditribute upline bonus flow
_uplineBonusDistribution()
buyerAddress -- address of a user for whom to distribute rewards.
processedAmount -- cost of limit pack.
isMatchingBonus -- flag which indicates matching rewards should be distributed.
isMatchingBonus -- flag which indicates matching rewards should be distributed.
Get user's 15 upper levels
contractTree.getLvlsUp (buyerAddress)
If isMatchingBonus is on, then activeIndex = 5 and activeLevel = 6. Otherwise activeIndex = 0 and activeLevel = 1.

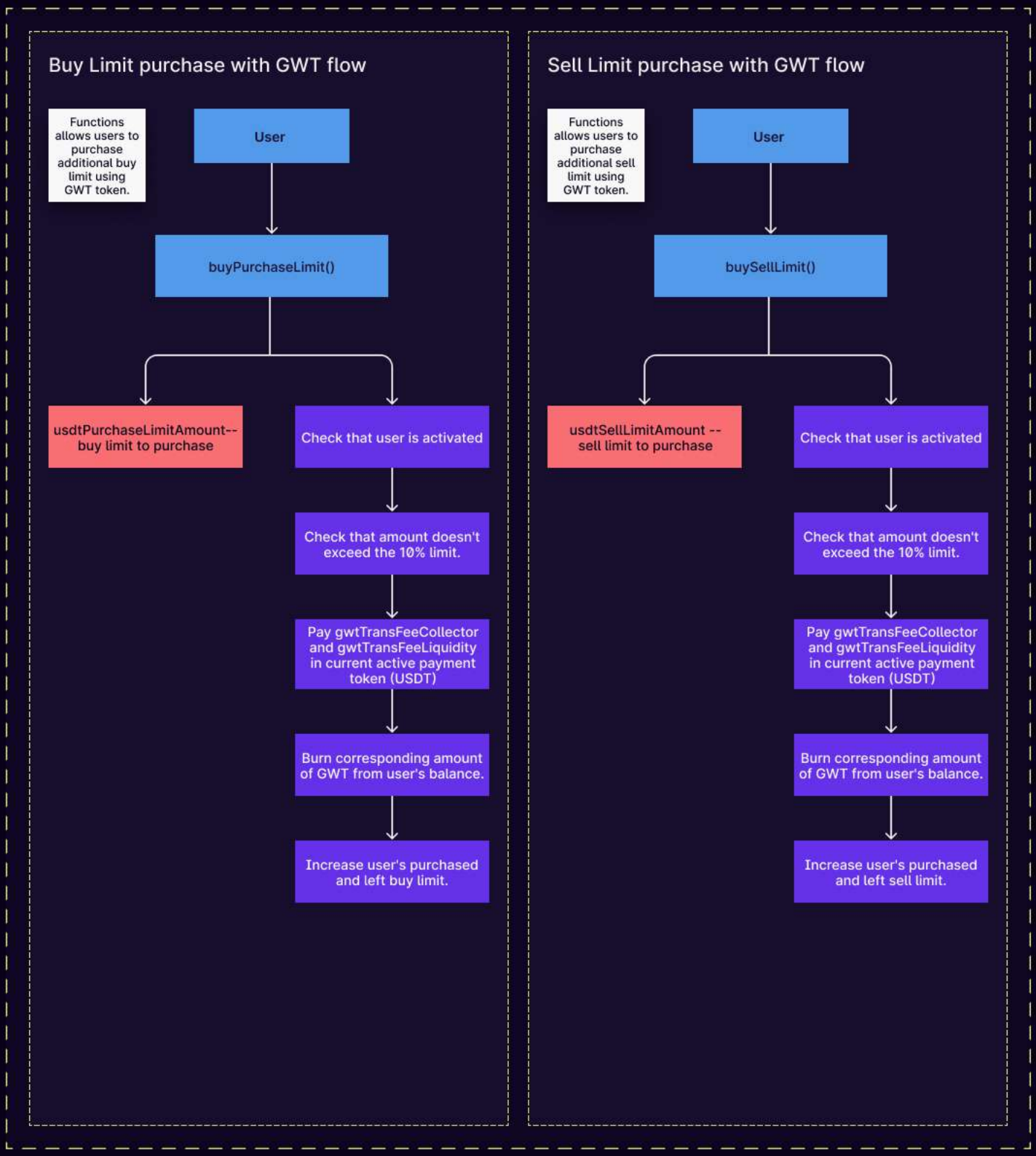

Ditribute upline bonus flow
Iterate through bonus distribution until it is completed.
The loop processes upline users of each index and level, thus distributing rewards through the tree of users.
Fetch new records for upline.
- Get uplineUsers for a user with index 14 from uplineUsers.
- Reset activeIndex.
Stop distribution if all 15 bonuses are distributed or next upline user is zero address.
Exit
If matching bonus mode is off, increase user's networkTurnover by packCost.
Increment activeIndex and activeLevel.
Rewards for this condition are calculated like: packCost/100* bonusPercent
Skip the iteration if user's earn limit at current level is equal to 0.
Increment activeIndex.
Level is open for current upline user and matching bonus mode is off.
Deposit bonus % of pack's cost to user's frozen balance.
matchingActive Until of user can be extended using the function extendMatching Bonus(). isMatchingBo nus is a boolean parameter of the function. It's false when buying a pack and true when withdrawing the balance.
Skip the iteration if matching bonus mode is on and user's matching bonus active status is expired.
Increment activeIndex.
Level is open for current upline user and matching bonus mode is on.
Deposit matchingBonus user's frozen balance.
Rewards for this condition are calculated like: packCost/100*tr eeLvls[activeLev el].bonus/100*tr eeLvls[activeLev el].frozenPercent
Skip the iteration if matching bonus mode is off and active leven is less than 4.
Increment activeIndex and activeLevel.
Level is closed for current upline user and matching bonus mode is on.
Deposit the current level frozen percent of matching rewards for user to user's frozen balance.
Level is closed for current upline user and matching bonus mode is off.
Deposit frozen percent of part of pack's cost to user's frozen balance.
Rewards for this condition are calculated like: matchingBonusAmount/100 *treeLvls[activeLevel].froze nPercent


Ditribute upline bonus flow
Matching bonus mode is on?
YES
NO
Return processedAmount-usedBonus
Return processedAmount/100*marketingDistributionFromPack-usedBonus

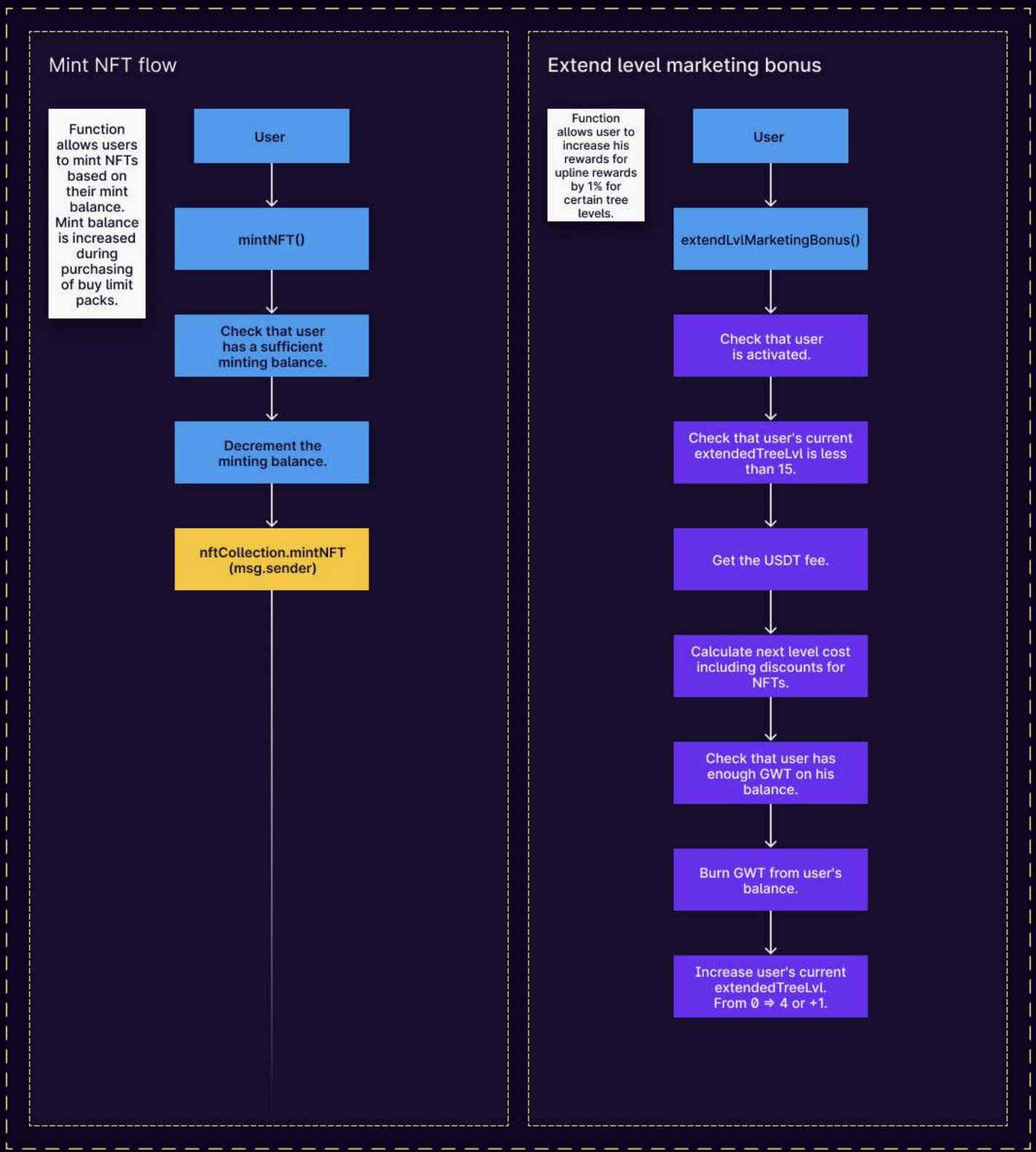

Unfreeze frozen funds flow
Function is used to release and funds frozen for user. Frozen tokens are added to the user's internal balance.
releaseFrozenFunds()
Function is called within several core functions, but can also be called directly by users.
releaseAddress -- address of user for whom funds are released.
releaseAddress -- address of user for whom funds are released.
Check that releaseAddress is not zero address.
Validate that paymentID is between 0 and 3.
For each level from 5 to 15.
Get user's frozen funds at current level.
Skip level if frozen funds equal 0.
If funds are overdue, send frozen funds to liquidityCollector instead.
Overdue is calculated based on timestamp when funds were frozen + freeze period of the level.
Check if max level of user's limit pack is greater than or equal to current level.
Add frozen funds to user's internal balance.
Amount of frozen funds which can be unfrozen depend on user's left earn limit. User can't unfrozen more than the limit. Unfrozen amount is deducted from the user's left earn limit.

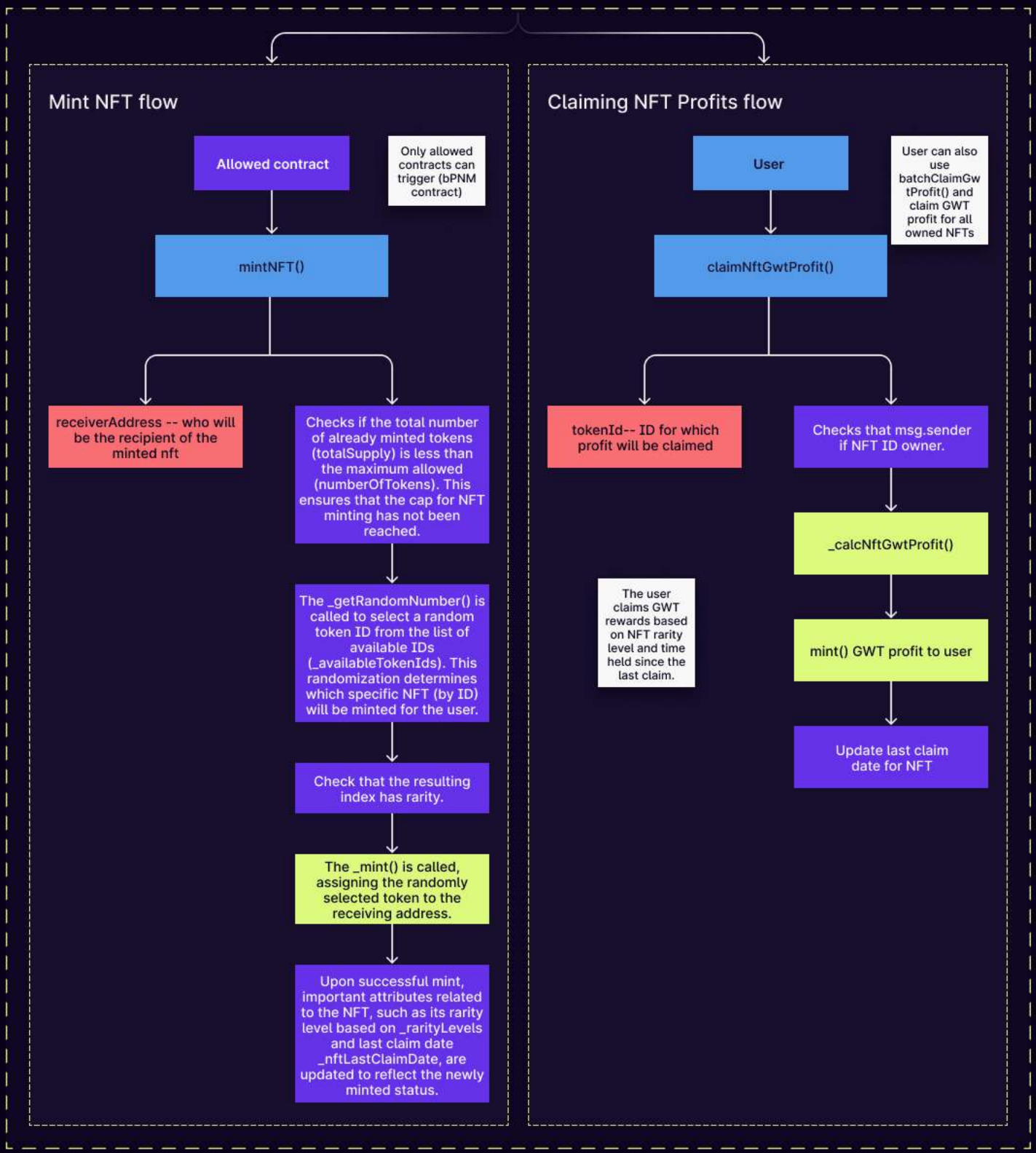

Extend Matching Bonus period
Extend the matching period bonus for user by 30 days. The bonus affects user's ability to collect matching bonus rewards during upline rewards distribution.
User
extendMatchingBonus()
Check that user is activated.
Check that user doesn't exceed the limit.
Calculate user's payment in GWT for the bonus taking in account discounts based on NTFs
Check that user has enough GWT on his balance.
Pay USDT fees.
Burn GWT from user's balance.
Extend user's matchingActiveUntil by matchingBonusExtendPeriod.

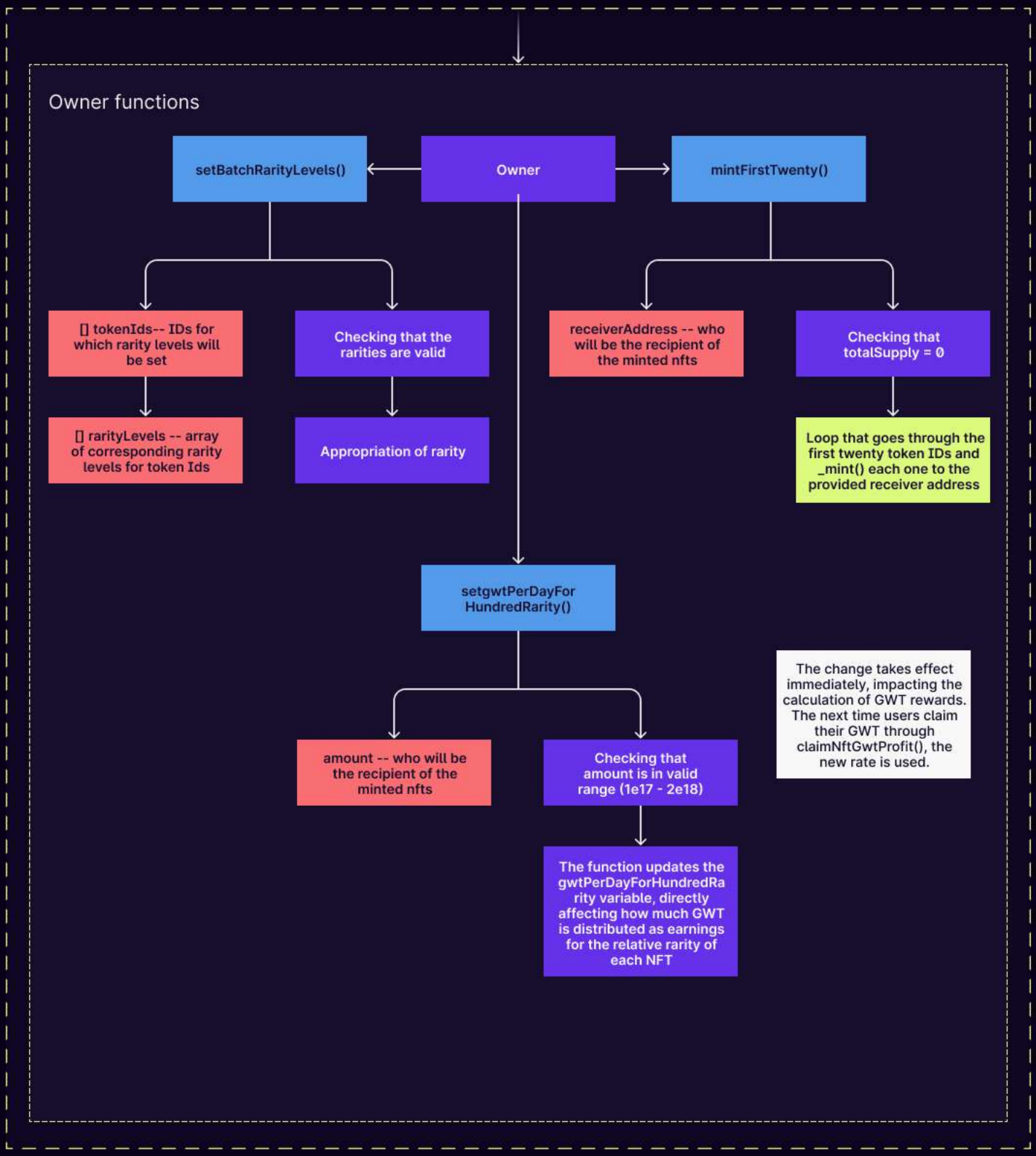

Earn limit purchase with GWT flow
Functions allows users to purchase additional earn limit using GWT token.
User
buyEarnLimitWithGwt()
earnLimitAmount -- earn limit to purchase
Check that user is activated
Check that user's current limitPackID >= 5
Check that amount doesn't exceed the 10% limit.
Pay gwtTransFeeCollector and gwtTransFeeLiquidity in current active payment token (USDT)
Burn corresponding amount of GWT from user's balance.
Increase user's purchased and left earn limit.
releaseFrozenFunds(msg.sender,_payment)


Buy Limit purchase with GWT flow
Functions allows users to purchase additional buy limit using GWT token.
User
buyPurchaseLimit()
usdtPurchaseLimitAmount-- buy limit to purchase
Check that user is activated
Check that amount doesn't exceed the 10% limit.
Pay gwtTransFeeCollector and gwtTransFeeLiquidity in current active payment token (USDT)
Burn corresponding amount of GWT from user's balance.
Increase user's purchased and left buy limit.
Sell Limit purchase with GWT flow
Functions allows users to purchase additional sell limit using GWT token.
User
buySellLimit()
usdtSellLimitAmount -- sell limit to purchase
Check that user is activated
Check that amount doesn't exceed the 10% limit.
Pay gwtTransFeeCollector and gwtTransFeeLiquidity in current active payment token (USDT)
Burn corresponding amount of GWT from user's balance.
Increase user's purchased and left sell limit.

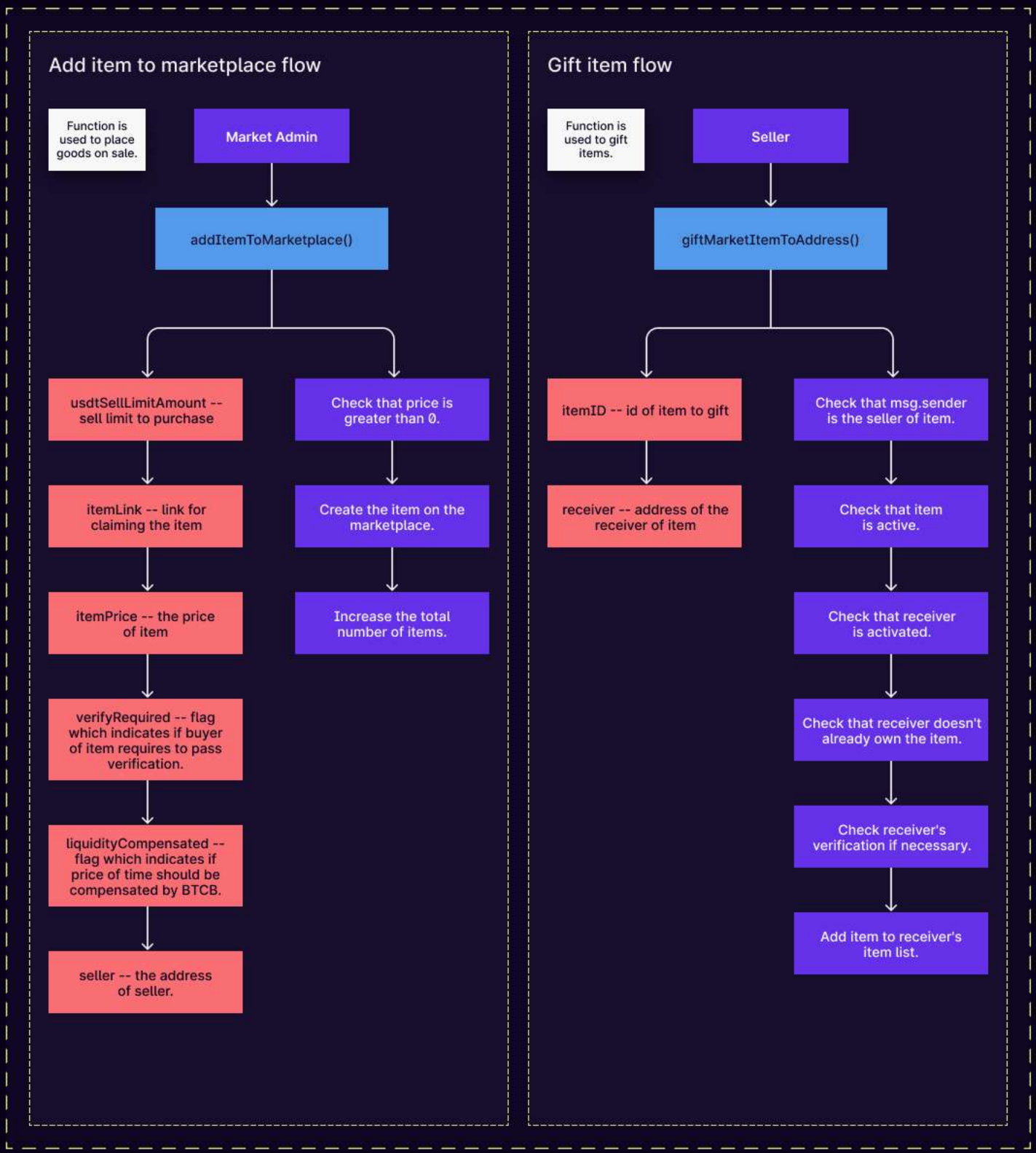
bPNM.sol

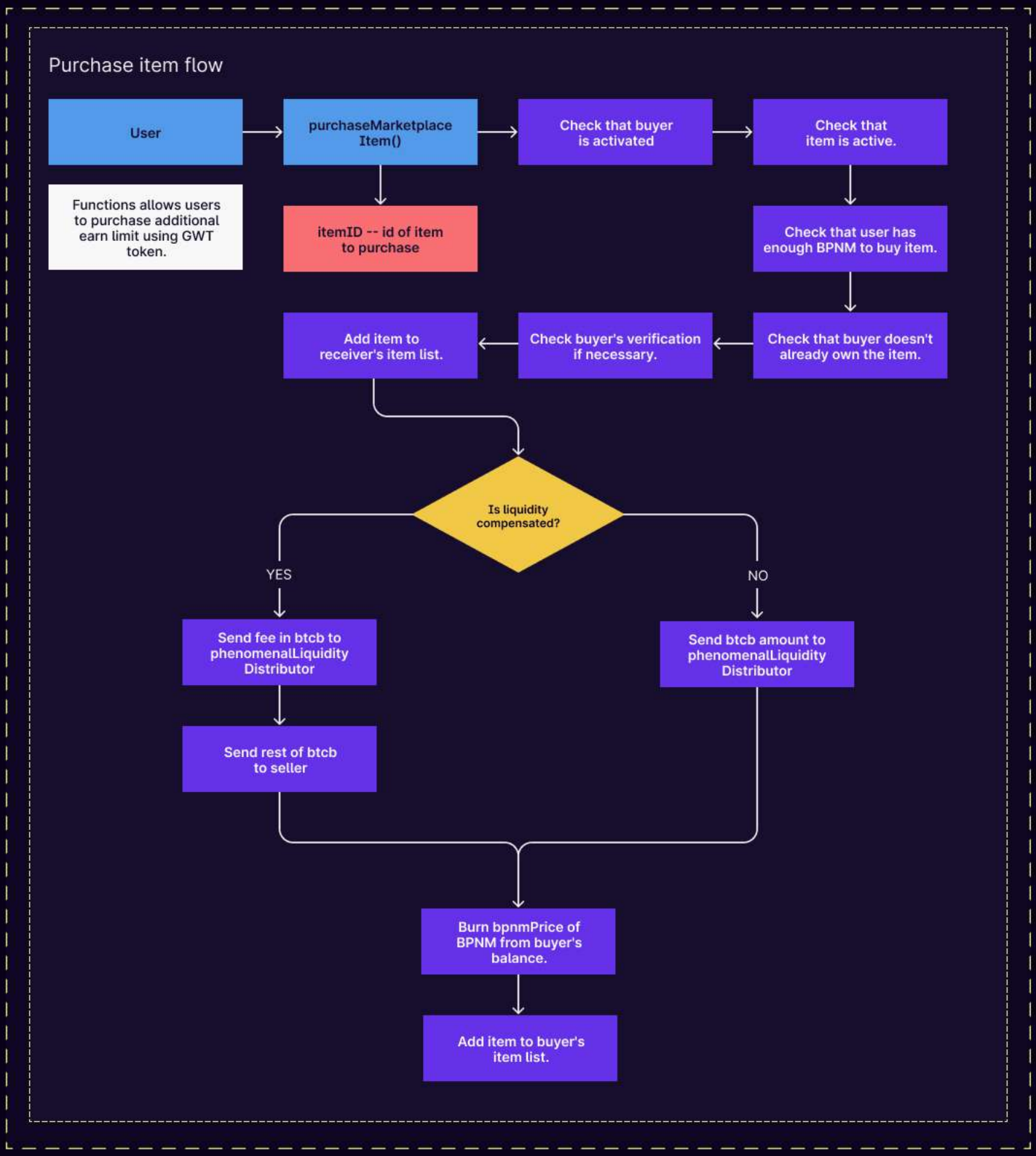

Mint NFT flow
Function allows users to mint NFTs based on their mint balance. Mint balance is increased during purchasing of buy limit packs.
User
mintNFT()
Check that user has a sufficient minting balance.
Decrement the minting balance.
nftCollection.mintNFT (msg.sender)
Extend level marketing bonus
Function allows user to increase his rewards for upline rewards by 1% for certain tree levels.
User
extendLvlMarketingBonus()
Check that user is activated.
Check that user's current extendedTreeLvl is less than 15.
Get the USDT fee.
Calculate next level cost including discounts for NFTs.
Check that user has enough GWT on his balance.
Burn GWT from user's balance.
Increase user's current extendedTreeLvl. From 0 ⇒ 4 or +1.


Mint NFT flow
Allowed contract
Only allowed contracts can trigger (bPNM contract)
mintNFT()
receiverAddress -- who will be the recipient of the minted nft
Checks if the total number of already minted tokens (totalSupply) is less than the maximum allowed (numberOfTokens). This ensures that the cap for NFT minting has not been reached.
The _getRandomNumber() is called to select a random token ID from the list of available IDs (_availableTokenIds). This randomization determines which specific NFT (by ID) will be minted for the user.
Check that the resulting index has rarity.
The _mint() is called, assigning the randomly selected token to the receiving address.
Upon successful mint, important attributes related to the NFT, such as its rarity level based on _rarityLevels and last claim date _nftLastClaimDate, are updated to reflect the newly minted status.
Claiming NFT Profits flow
User
User can also use batchClaimGwtProfit() and claim GWT profit for all owned NFTs
claimNftGwtProfit()
tokenId-- ID for which profit will be claimed
Checks that msg.sender if NFT ID owner.
_calcNftGwtProfit()
The user claims GWT rewards based on NFT rarity level and time held since the last claim.
mint() GWT profit to user
Update last claim date for NFT


Owner functions
setBatchRarityLevels()
Owner
mintFirstTwenty()
[] tokenIds-- IDs for which rarity levels will be set
Checking that the rarities are valid
receiverAddress -- who will be the recipient of the minted nfts
Checking that totalSupply = 0
[] rarityLevels -- array of corresponding rarity levels for token Ids
Appropriation of rarity
Loop that goes through the first twenty token IDs and _mint() each one to the provided receiver address
setgwtPerDayFor HundredRarity()
The change takes effect immediately, impacting the calculation of GWT rewards. The next time users claim their GWT through claimNftGwtProfit(), the new rate is used.
amount -- who will be the recipient of the minted nfts
Checking that amount is in valid range (1e17 - 2e18)
The function updates the gwtPerDayForHundredRarity variable, directly affecting how much GWT is distributed as earnings for the relative rarity of each NFT

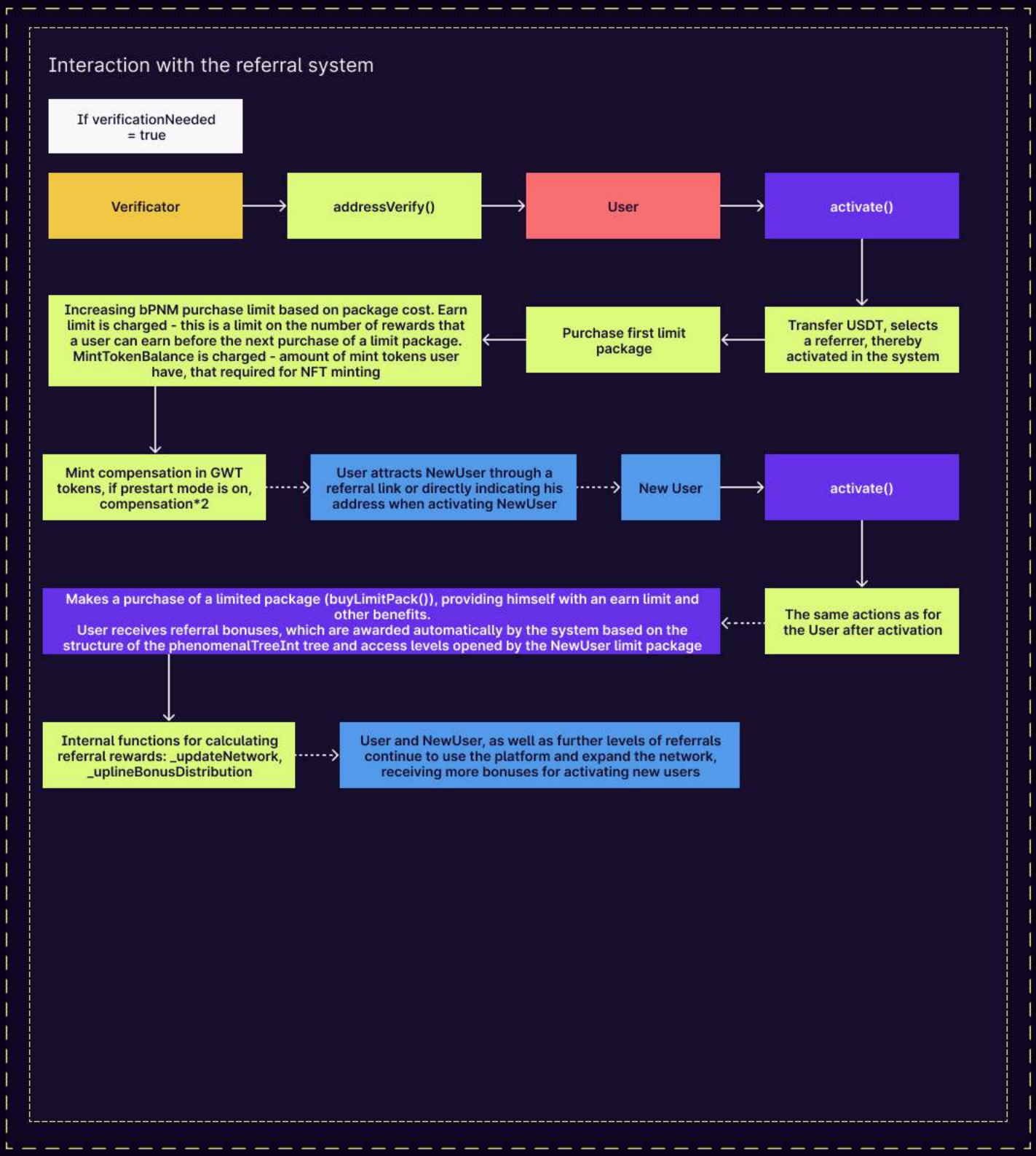
bPNM.sol


Add item to marketplace flow
Function is used to place goods on sale.
Market Admin
addItemToMarketplace()
usdtSellLimitAmount -- sell limit to purchase
itemLink -- link for claiming the item
itemPrice -- the price of item
verifyRequired -- flag which indicates if buyer of item requires to pass verification.
liquidityCompensated -- flag which indicates if price of time should be compensated by BTCB.
seller -- the address of seller.
Check that price is greater than 0.
Create the item on the marketplace.
Increase the total number of items.
Gift item flow
Function is used to gift items.
Seller
giftMarketItemToAddress()
itemID -- id of item to gift
receiver -- address of the receiver of item
Check that msg.sender is the seller of item.
Check that item is active.
Check that receiver is activated.
Check that receiver doesn't already own the item.
Check receiver's verification if necessary.
Add item to receiver's item list.

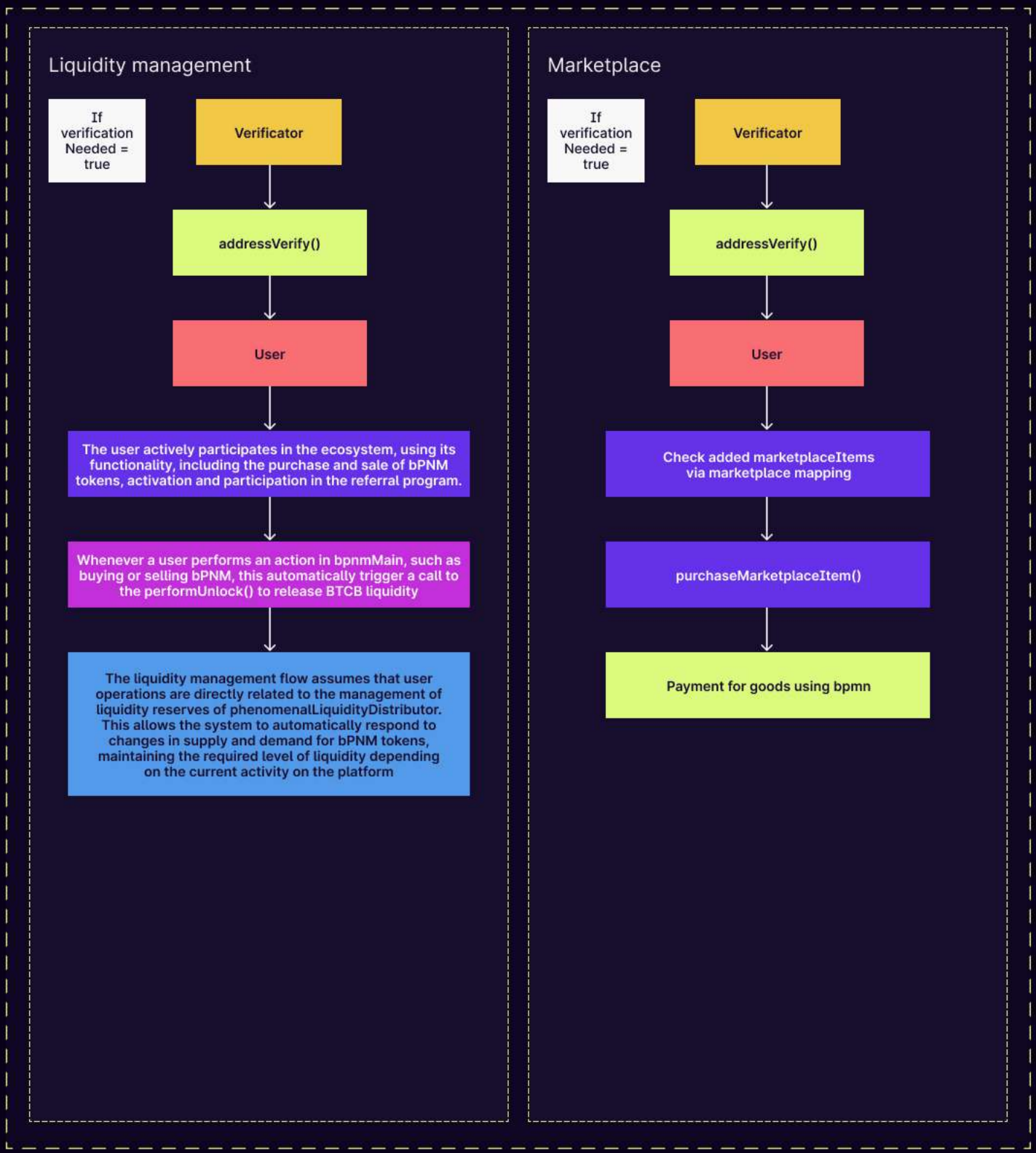

Purchase item flow
User
purchaseMarketplace Item()
Check that buyer is activated
Check that item is active.
Functions allows users to purchase additional earn limit using GWT token.
itemID -- id of item to purchase
Check that user has enough BPNM to buy item.
Add item to receiver's item list.
Check buyer's verification if necessary.
Check that buyer doesn't already own the item.
Is liquidity compensated?
YES
NO
Send fee in btcb to phenomenalLiquidity Distributor
Send btcb amount to phenomenalLiquidity Distributor
Send rest of btcb to seller
Burn bpnmPrice of BPNM from buyer's balance.
Add item to buyer's item list.

Possible general flows

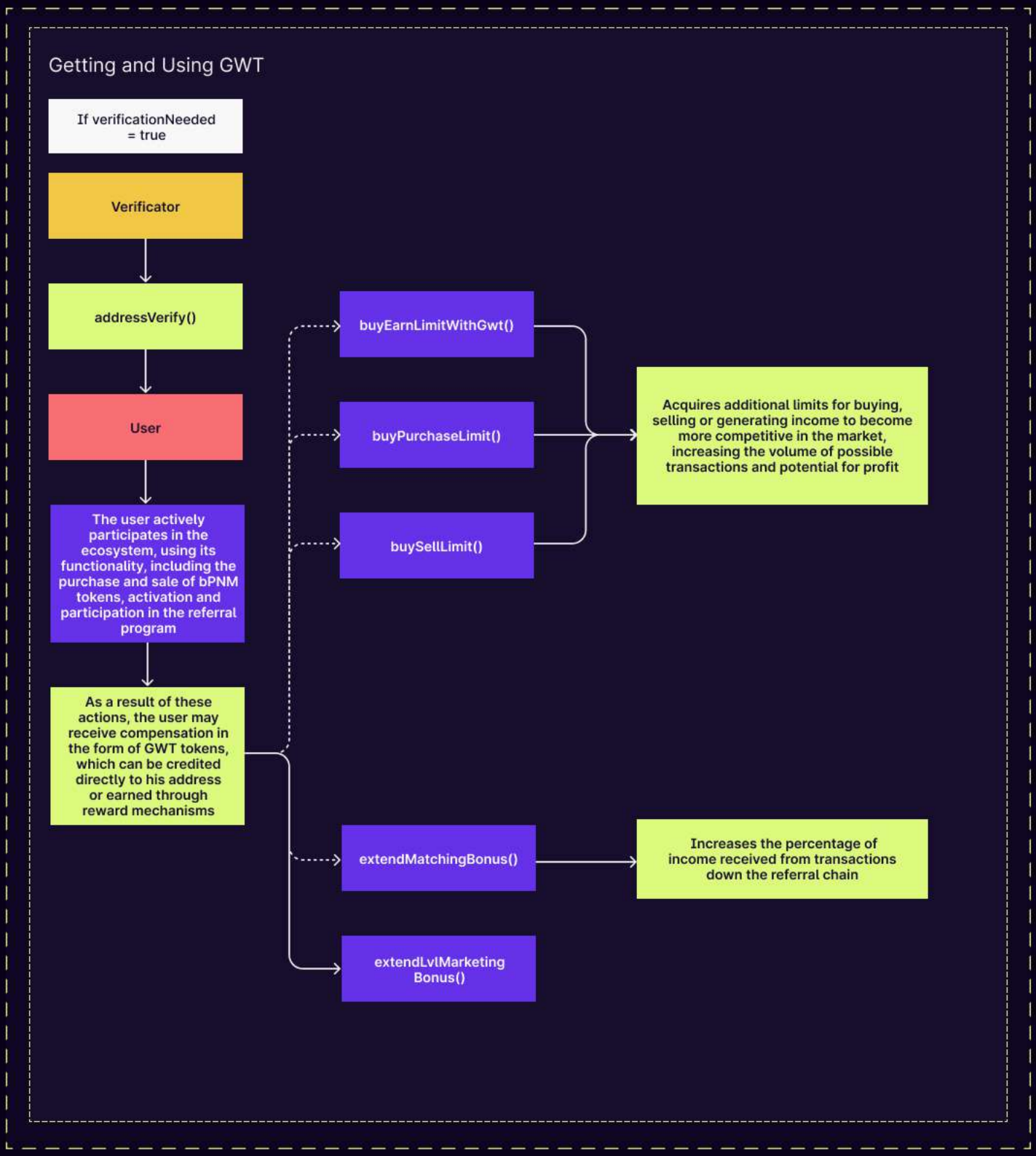
Possible general flows



Possible general flows



Possible general flows



BPNM DESCRIPTION

Description

1. **bpnmMain.sol:**

- This contract acts as the core of the bPNM token system, based on a modified ERC20 standard, and includes various functionalities such as initial address activation, limit pack purchases, and a marketplace.
- It includes integrations with multiple interfaces and smart contracts, such as payment methods, a BTC/USD price oracle, NFT consultants, and Phenomenal tree structures for networking
- The contract exhibits mechanisms for token purchase limitations, differentiation of payment methods, bonus and fee distribution, tree-based frozen fund management, and marketplace
- It strongly emphasizes access control and state management to ensure only verified and activated users can execute certain
- Prestart and lock modes regulate the accessibility of functions according to different phases of the project's lifecycle.

2. **PhenomenalConsultants.sol:**

- Serving as an NFT collection, this contract allows users to mint, own, and transfer NFTs, which provide utility within the DNPM ecosystem in the form of discounts and passive GWT farming. It introduces a rarity system that affects the NFT's value regarding discounts and GWT yield. Once set, the rarity logic is immutable, and pre-minted NFTs have special characteristics.
- Tokens are uniquely identifiable via IPFS-stored metadata and images. The minting process uses a non-Chainlink-based randomness generator factoring in block variables and a nonce.
- Transfers of NFTs require a fixed USDT fee, and the contract controls GWT profit claims for NFT owners based on owned rarity and last claim date.

3. **phenomenalLiquidityDistributor.sol (PLD):**

- The PLD contract is a standalone liquidity management tool for the bPNM ecosystem, focusing on unlocking and distributing BTCB liquidity to the bPNM contract.
- It features a timing and percentage-based unlock system, influenced by the balances within bPNM itself, and a stable payment contract tied together through a performance formula.
- Prestart mode indicates that the contract is not active yet, with liquidity unlock functionalities becoming operational after exiting this stage.
- The liquidity unlock operation is triggerable by the bpnmMain contract events when performing token-related actions (e.g. buy, sell, pack purchase).

BPNM DESCRIPTION

Roles and Responsibilities

1. **Owner:**

- The owner has the highest administrative privileges and complete contract control.
- Capable of initiating contract updates.
- Manages the assignment of roles and the withdrawal of rights.

2. **Promoter:**

- Manages contract parameters such as fees, feature toggles, and operational thresholds across various contracts.
- In the PhenomenalConsultants contract, the promoter can change the GWT payout percentage for NFTs (see function setgwtPerDayForHundredRarity).
- In the phenomenalLiquidityDistributor contract, the promoter can modify the liquidity unlock period and percentage (see functions setUnlockPeriod and setUnlockPercent) and disable the prestart mode.
- In the bpnmMain contract, the promoter adjusts critical financial parameters such as multipliers for buy limits, compensation percentages, and additional GWT-related costs (see functions like setBuyLimitMultiplier, setSellLimitMultiplier, setLimitPackPurchaseGwtCompensation, etc.).

3. **Marketplace Administrator:**

- Authorized to manage marketplace items in the bpnmMain contract, such as adding or removing products from the marketplace.

4. **Verificator:**

- Responsible for user verification, validating their activation, and access to certain contract functions (see addressVerify function and onlyVerified modifier).

5. **Fee Collector:**

- Intended for collecting fees accumulated in contract operations (see feeCollector).

6. **Liquidity Collector:**

- Derives funds from various streams within the ecosystem, such as transaction fees, token sales, service charges, and other financial operations that require liquidity pooling (see liquidityCollector in bpnmMain).
- Collect funds for further distribution or use for certain liquidity operations (see liquidityCollector in phenomenalLiquidityDistributor.sol).

7. **General User:**

- Any individual or entity that directly interacts with the contract.
- Users can activate addresses, purchase limited packs in bpnmMain, or mine and manage NFTs in PhenomenalConsultants.
- To fully utilize the platform, a user must first become recognized within the system by undergoing an activation process. This process typically involves an initial account “activation" that might necessitate purchasing a start-up package or being referred by an existing user.

List of values assets

1. **bPNM Tokens (a modification of ERC20):**

- The primary asset managed within the bpnmMain contract, bPNM tokens are crucial for platform operations, including limit pack purchases, market interactions, and partaking in the bonus and reward system.

2. **Collectible NFTs (ERC721):**

- In PhenomenalConsultants, the contract oversees collectible NFTs, which offer additional benefits such as discounts within the DNPM ecosystem and passive GWT earnings.

3. **Collected Transaction Fees:**

- Fees configured and collected for various transactions in bpnmMain add significant value and represent a revenue source for the contract owners or designated fee collectors.

4. **BTCB Liquidity (ERC20):**

- The phenomenalLiquidityDistributor manages BTCB liquidity which can be unlocked and transferred to bpnmMain to maintain stability and facilitate transactions within the platform.

5. **Funds in the Role of Liquidity Collector:**

- Funds accumulated at the `liquidityCollector` address form liquidity reserves, upholding the financial system of DNPM and its operations.

Settings

1. Transaction Fee Settings: Set and adjust the fee structure for buying and selling bPNM tokens, along with other system usage charges.
2. Limit Multiplier Settings: Define the multipliers for buying and selling limits of bPNM.
3. Payment Token Addresses: Select the tokens that are accepted as means of payment.
4. Referral Bonus Settings: Manage the size and distribution of referral bonuses.
5. Referral Tree Settings: Configuration of referral system structure.
6. Price Oracle Selection: Binance and ChainLink oracles for current BTC pricing.
7. BTCB Liquidity Unlock: Parameters for the period and percentage of BTCB liquidity unlocking.
8. NFT Metadata URI: Configuration of the base URI for NFT metadata.
9. Prestart Mode: A parameter that defines whether the system is in pre-launch mode.
10. Roles Permissions and Restrictions: Set and adjust the available functionalities for different roles within the system.
11. GWT Compensation Settings: Configuration rules for determining the amount of GWT compensation.
12. Pack Sale Distribution Percentage: The percentage of revenue from limit pack sales directed towards marketing rewards and other purposes.
13. NFT Discount Settings: Set the discount rate for users when purchasing based on the rarity of owned NFTs.
14. Lock and Unlock Functions: Manage the system lock state to enable and disable functionality.

Deployment script

Deployment scripts weren't provided.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

MEDIUM-1 | VERIFIED

Inconsistent Payment Token Handling.

bpnmMain.sol: _replenishPaymentBalance.
The _replenishPaymentBalance function facilitates payment for packs potentially called within the activate() function. Currently, the cost of packs is predefined, presumably in USDT. If the payment method changes in the future, the cost is not automatically recalibrated to match the new token's value, leading to discrepancies and possibly incorrect payment amounts.

Recommendation:

Implement a mechanism that adjusts the pack costs dynamically based on the active payment token. This could involve real-time price feeds or an exchange rate system to maintain consistent pricing across accepted tokens. Otherwise, verify if the price and decimals of all payment tokens will be equal (for example, they are stablecoins).

Post-audit:

The team has acknowledged the issue and assured that only stable coins with the same decimals will be used (USDT, DAI, etc). Also, initially, only the USDT will be used, and any other added payment options will be carefully double-checked to correspond to the requirements.

LOW-1 | VERIFIED

Arbitrary User Activation and Referral Reward Capture.

bpnmMain.sol: activate().
The activate() function in the contract allows for a new user registration without strict validation of its origin. This raises the potential issue that any actor can register a new user and set themselves as the referrer, thus illegally positioning themselves to receive referral rewards.

Recommendation:

Implement additional checks to ensure that the new user genuinely intends to be referred by the caller. This could include a confirmation mechanism from the new user or a secure off-chain process that ensures new user consent before being added to the system with an associated referrer.

Post-audit:

The team has verified that the functionality is used to simplify the onboarding of new users who are not familiar with Web3. Additionally, abusing the issue won't grant too many benefits which is why it is initially marked as low.

INFO-1 | RESOLVED

Lack of Event Emission.

Most of the functions in the contracts are missing emitting events, although they change the state of the contract.
To keep track of historical changes in storage variables, it is recommended that events be emitted for every change in the functions that modify the storage.

Recommendation:

Consider emitting events in all functions where state changes to reflect important changes in contracts.

INFO-2 | VERIFIED

Risk of Tokens Getting Stuck When Sent to Contract Unintentionally.

Based on the nature and purpose of the contract, the team should consider the case that Users might accidentally (or purposely, after misreading the flow and dApp interface) send ERC-20 or ERC-721 tokens directly to the contract's balance, resulting in a loss of access to these assets.

The issue is marked as Info, as it is related to the expected user behavior rather than contract functioning. So, the possible issue should be listed in the report and require feedback from the team, though the “rescue” mechanism is not mandatory.

Recommendation:

Verify that the dApp will have clear instructions on how users should interact with the protocol. Consider implementing a “rescue” mechanism that permits the contract owner or authorized individuals to retrieve tokens accidentally sent to the contract - though with the appropriate security checks regarding the balance stored via the legitimate flow.

Post-audit:

The team has verified that interaction with the code will only happen via the dashboard. However, we suggest putting a note to notify users not to send funds directly.

Violations of Solidity Style Guide Conventions - magic numbers.

Hard-coded constants like:

bpnmMain: setNftMintTokenMaxAmount(), line 1769 → 3000, 10 000;
_buyLimitPack() → 1e6, 1200, 10, 100;
getBtcPrice() → 60*60*4;
_uplineBonusDistribution(), line 802 → 10; after line 843 → 100;
_depositBonusToUser(), line 923 → 100*90;
_depositFrozenBonusToUser() → 100*90;
buyBpnm() → 100, 10;
sellBpnm() → 100;
withdrawBalance() → 100, 1200;
extendMatchingBonus() → 1200, 100*30;
buyEarnLimitWithGwt() → 100;
buyPurchaseLimit() → 100;
buySellLimit() → 100;
extendLvlMarketingBonus() → 1200, 10;
PhenomenalConsultants: _calcNftGwtProfit() → 86400, 100;
phenomenalLiquidityDistributor: performUnlock() → 10000;
setUnlockPeriod() → 1 hours.

are not self-explanatory or defined as named constants, making it difficult to understand their meaning or how they were derived.

Recommendation:

Replace magic numbers with named constants that provide context and make the code more maintainable and understandable.

Post-audit:

The team has postponed fixing the issue since it is not meant to be public.

INFO-3.2 | RESOLVED

Violations of Solidity Style Guide Conventions - filename and contract name mismatch.

bpnmMain - bpnm;
PhenomenalConsultants - NFT_consultants;
phenomenalLiquidityDistributor - phenomLiquidityDistributor.
There is an inconsistency between the filenames and the declared contract names within the file, which can lead to confusion when maintaining the codebase or conducting further audits.

Recommendation:

Rename the Solidity file to exactly match the contract name inside the file to adhere to standard naming conventions and improve code maintainability.

Post-audit:

New files for contracts were added, which match with contracts' names.

INFO-3.3 | ACKNOWLEDGED

Violations of Solidity Style Guide Conventions - confusion naming.

The presence of variables like `value` and `_value` within the same codebase could cause confusion.

Recommendation:

Align variable naming with the style guide, ensuring that parameters are named clearly without prefixes or postfix underscores unless indicating scope as per the style guide.

Post-audit:

Variable naming wasn't aligned.

INFO-3.4 | UNRESOLVED

Violations of Solidity Style Guide Conventions - Event and Struct names.

bpnmMain.sol, PhenomenalConsultants.sol, phenomenalLiquidityDistributor.sol: Various Events and Structs.
Some events and structs start with a lowercase letter, contrary to the practice of using uppercase initials to enhance readability and signal their roles as user-defined types.

Recommendation:

Refactor the affected event and struct names to start with uppercase letters.

Post-audit:

Events were renamed. However, the names of some structs still start with lowercase or underscore.

INFO-4 | VERIFIED

The owner is able to set the URI.

PhenomenalConsultants.sol: setBaseURI().
The base URI is the core value of NFTs, which references the images of tokens stored in the decentralized storage. Since users usually pay to own the image, the base URI remains immutable to ensure that users won't lose ownership of the image. However, the function setBaseURI() allows the contract owner to change this value, potentially changing the images owned by users.

Recommendation:

Verify if the functionality of updating the URI is necessary for the collection. Notify users in advance if the URI is going to be changed.

Post-audit:

The team has verified that the setter is necessary in case the NFT URI is swiped from IPFS. Additionally, NFT's main value is its immutable rarity level.

Lack of Validation.

PhenomenalConsultants.sol: constructor() → parameters _usdtTokenAddress, _feeCollector, _gwt, _numberOfTokens;

phenomenalLiquidityDistributor: constructor() → parameters _btcbTokenAddress, _bpnmTokenAddress;

bpnmMain: init() → parameters _depositTokenAddress, _btcbTokenAddress, _contractTree, _nftCollection, _gwt, collectorBTCB.

The zero check is a standard validation to prevent initializing contracts without valid addresses and amounts. Before setting them, add necessary checks to ensure that variables that should not be `0` and none of the addresses are equal to the zero address.

Recommendation:

Consider adding the necessary validation.

INFO-6 | RESOLVED

Unclear External Interaction.

phenomenalLiquidityDistributor.sol: performUnlock().
The contract comments in the performUnlock() suggest that it should be called from the bpnmMain contract. However, there is no explicit enforcement within the code, such as a modifier or a require statement, to ensure that it is only callable by the bpnmMain contract. This could lead to confusion or unintended usage, as any address could invoke this function.

Recommendation:

If the intention is to restrict the function to being called only by the bpnmMain contract, introduce a modifier that checks the msg.sender against the authorized contract's address. Alternatively, if the function is meant to be public and callable by any address, update the comments to reflect this to avoid ambiguity.

Post-audit:

A comment was added explaining that anyone can call the function.

INFO-7 | RESOLVED

Redundant Calculations.

bpnmMain.sol: _buyLimitPack(), lines 639, 641.
There are lines within the function where calculations are repeated unnecessarily, which can lead to inefficiencies in contract execution.

Recommendation:

Optimize the function by storing the calculation result in a temporary variable and reusing it in subsequent operations to reduce redundant processing and gas costs.

INFO-8 | ACKNOWLEDGED

Testnet Code Remnant in Production Smart Contract.

bpnmMain.sol: getBtcPrice().
The function contains hardcoded values intended for testnet use, which is noticeable in pre-production smart contract code. While convenient during the development and testing phases, such code should be removed or replaced with production-ready mechanisms before deployment.

Recommendation:

Ensure that the testnet-specific code is removed before deployment.

INFO-9.1 | VERIFIED

Potential Optimization - Unnecessary Token Balance Checks.

bpnmMain.sol: buyBpnm(), withdrawBalance(), extendMatchingBonus().
Several functions perform explicit balance checks for a user's token holdings, as the ERC20 `transfer` and `transferFrom` functions revert the transaction if the balance is insufficient. These preliminary checks could be removed to simplify the code and save on gas costs without sacrificing security.

Recommendation:

Remove the redundant token balance validations in functions.
Post-audit. The balance checks are necessary for easier transaction analysis. Since transactions on BSC don't cost much, leaving the checks will not significantly increase transaction costs.

INFO-9.2 | RESOLVED

Potential Optimization - by Caching Value.

bpnmMain.sol: bpnmPrice().
The value calculated by the bpnmPrice() function could be reused within transactional operations instead of recalculating multiple times. Currently, the function is called repeatedly in different parts of the contract, which may result in unnecessary gas spending.

Recommendation:

Cache the result of bpnmPrice() in a memory variable during transactions where multiple price lookups occur.

INFO-9.3 | RESOLVED

Potential Optimization - Conditional Checks.

bpnmMain.sol: _uplineBonusDistribution().
The function's series of if statements iterate through upline user bonus calculations without utilizing an `if-else` structure. This leads to multiple unnecessary checks even after a matching condition is found, leading to suboptimal gas usage.

Recommendation:

Refactor the _uplineBonusDistribution() function to employ an `if-else` to break out of the conditional checks once a matching condition is met.

INFO-9.4 | RESOLVED

Potential Optimization - Unnecessary Validation Checks for Unsigned Integer Parameters.

Several functions throughout the bpnmMain contract verify that the unsigned integer parameters are greater than or equal to zero. This is unnecessary as the uint256 datatype inherently guarantees that the value cannot be negative. This tautological condition results in redundant code and gas wastage during execution.
Examples of this issue are prevalent in functions such as _getPaymentContract, releaseFrozenFunds, addressFrozenTotal, setBtcOracle, setBpnmBuyFee, setBpnmSellFee, and setgwtTransFeeLiquidity, all of which contain the unnecessary comparison against zero for uint256 parameters.

Recommendation:

Remove checks like this `paymentID >= 0` from the required statements in the respective functions.

INFO-9.5 | RESOLVED

Potential Optimization - Inefficient State Variable Declaration (not immutable).

phenomenalLiquidityDistributor: btcb, bpnm;
PhenomenalConsultants: feeCollector, gwt, numberOfTokens, usdt.
The contracts define state variables set during contract deployment and do not change afterward. However, these variables are not currently marked as immutable, meaning they can be more gas-intensive to access than necessary.

Recommendation:

Mark described state variables with the immutable keyword where they are declared.

INFO-9.6 | RESOLVED

Potential Optimization - Inefficient State Variable Declaration (not constant).

bpnmMain: _binanceBtcOracle, _chainLinkBtcOracle.
Described addresses are set at deployment and are intended to remain unchanged for the contract's lifetime. Not declaring these as constant results in unnecessary gas costs every time they are accessed.

Recommendation:

If the intent is for _binanceBtcOracle and _chainLinkBtcOracle to remain static and not to be updated post-deployment, mark these variables as constant.

INFO-10 | VERIFIED

Incentive Misalignment in Maximum Pack Purchases.

bpnmMain.sol: _buyLimitPack().
Currently, purchasing the maximum available limit pack seems less profitable than buying smaller packs sequentially due to the lack of cumulative bonuses from previous packs. Additionally, the system does not allow downgrading the pack ID, which may discourage purchasing larger packs upfront.

Recommendation:

Adjust the bonus system to account for cumulative rewards when users purchase larger packs or introduce the ability to downgrade packID, providing more flexibility and fair incentives to users at all investment levels.

Post-audit:

The team has verified that high-limit packs grant access to lower tree levels, so the profit is significantly bigger when getting bonuses from lower tree levels. Getting people on that level takes some time, so lower packages give more profits at the beginning, but later, when tree level depth grows, higher packages give more profit while decreasing limits to increase the flow of liquidity for token price growth.

INFO-11 | VERIFIED

Storage of Claim Links for Purchased Items.

bpnmMain.sol.
The contract stores the claim links for purchased marketplace items in the smart contract storage, which raises accessibility concerns. Given that these links are stored on-chain, they are publicly visible, and anyone can potentially access the URLs without needing to execute the purchase function and without validating ownership of the related item.

Recommendation:

Verify the necessity of storing claim links directly within the blockchain. If the links must stay private or only accessible to legitimate purchasers, consider alternative approaches.

Post-audit:

The front end will use an authorization layer via wallet to verify that the user has bought the item.

INFO-12 | ACKNOWLEDGED

Custom errors should be used.

Starting from the 0.8.4 version of Solidity it is recommended to use custom errors instead of storing error message strings in storage and use “require” statements. Using custom errors is more efficient regarding gas spending and increases code readability.

Recommendation:

Use custom errors.

Post-audit:

The team has acknowledged the issue, and since most of Dapp is already implemented, custom errors will not be implemented.

INFO-13 | RESOLVED

Inefficient Use of Arithmetic Operations.

Several times throughout the smart contracts, division operations are immediately followed by multiplication within the same formula. This ordering can sometimes lead to unnecessary precision loss due to Solidity's truncation behavior when handling integer divisions. This is a concern in financial calculations where every currency unit is significant.

Recommendation:

Review the arithmetic order of operations in financial formulas throughout the smart contracts. Consider restructuring calculations to perform multiplication before dividing to minimize precision loss. For example, adjust `a / b * c` to `a * c / b`, ensuring the multiplication does not overflow the variable's capacity.

Post-audit:

Calculations were restructured everywhere except:
BEP20BPNM.sol: buyBpnm() → lines 1059, 1061;
PhenomenalLiquidityDistributor.sol: performUnlock() → line 79.

INFO-14 | RESOLVED

Redundant Contract Copies in the Repository.

A recent review of the fixed contracts found that multiple copies of the same contract were left in the repository. This may confuse determining the current, active version of any given contract.

Recommendation:

Remove outdated or redundant contract copies from the repository.

INFO-15 | VERIFIED

Mismatch in Calculation Compared to Comment Description.

bpnmMain.sol: _uplineBonusDistribution(), line 800.
The code within the function divides the processed amount by 10, which implies that each address should receive 10% of the bonus pool. However, the comment indicates that each address is supposed to receive 0.5% from a 5% bonus pool. Based on the comment's description, the current implementation would result in each address receiving a significantly higher bonus than intended.

Recommendation:

Align the matching bonus distribution logic within the function to match the percentage described in the comments.
Post-audit. The team has verified that the functionality is working as intended.

INFO-16 | VERIFIED

Multiple Purchases of the Same Marketplace Item by Different Users.

bpnmMain.sol: purchaseMarketplaceItem().
The function allows multiple users to purchase the same item from the marketplace. This could be either intentional, allowing limitless sales of a digital product, or an oversight if the item should be unique or have limited availability.

Recommendation:

Verify if the ability for numerous users to purchase the same item aligns with the intended marketplace logic. If items are meant to be unique or have limited quantities, implement a tracking system to decrease the available quantity upon each purchase or restrict further purchases once sold out.

Post-audit:

According to the team, this is intentional functionality. If an item is out of stock, the seller will have to disable it.

INFO-17 VERIFIED

Inconsistencies with the whitepaper.

1. For the 4th tree level in the whitepaper, the following values are set: Percentage reserved from bonus size - 80, and Maximum reserve period in days - 2. But in the bpnmMain contract the set values for these items are 0, 0.
2. In bpnmMain matchingBonusGwtCost is set to 200 by default, although 500 is indicated in the whitepaper.
3. For nftDiscountForWithdraw, the contract in the commentary indicates a maximum of 5%, although, in the whitepaper, it is 50%.

Note:

This issue is marked as info because the discrepancies do not threaten the contract's immediate functionality or security. However, they indicate potential misalignments between the project's documentation and its technical implementation, which could lead to confusion among users and stakeholders.

Recommendation:

Review and update contract values to accurately reflect the project's intended design as per the whitepaper, or vice versa, to maintain consistency and clear expectations.

Post-audit:

The team has verified the issue so that:

1. Mistake in whitepaper. Initially, These settings were intended but then realized that lvl4 is opened on the lowest limit pack, so funds can never be frozen there. Will remove the numbers for the 4 levels from the whitepaper.
2. This is because the default value would be 500, but at the beginning, prestart is activated, which would be 200 at prestart. Later, it will be increased to 500 to match the whitepaper number.
3. This depends on what amount to calculate. In contrast, 5% means the absolute withdrawal amount, which is more correct for developers to understand. In the whitepaper, it is stated as 50% of the applied fee (because the fee is 10% of the absolute amount), which is easier for users to understand. So, basically, the max fee is 10% of the absolute amount, from which half can be compensated with NFT ownership.

bpnmMain.sol | PhenomenalConsultants.sol | phenomenalLiquidityDistributor.sol	
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by bPNM team

As a part of our work assisting bPNM in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the bPNM contracts requirements for details about issuance amounts and how the system handles these.

Scope:
bpnmMain.sol
PhenomenalConsultants.sol
phenomenalLiquidityDistributor.sol

dPNM
  ==1) Deployment bPNM
    ✓ Limit packs should be inited correctly (57507ms)
    ✓ Tree levels bonus percenta should be set correct (91ms)
    ✓ Zero user should be initiated
    ✓ First user should be activated (252ms)
  ==2) First user activating
    ✓ Should exist (107ms)
    ✓ Have limit pack
    ✓ Prestart GWT deposited
    ✓ Earn limit increased
    ✓ USDT liquidity increased for 8 USDT, fee collector earned 2 USDT (42ms)
  ==3) Packs purchase. 16 users activated
    ✓ Bonus goes for 15 lvls up (5257ms)
    ✓ Second pack purchase (1413ms)
    ✓ First user waits until frozen timer expires (1285ms)
    ✓ Compression check (3664ms)
    ✓ Pack autopurchase check (3674ms)
    ✓ +1% to marketing check (2038ms)
    ✓ Frozen unlocked to liquidity on timer finish (4961ms)
    ✓ Earn limit covers part of frozen bonus (1907ms)

✓ New frozen bonus move old frozen to liquidity on timer fininshed (1762ms)
==4) Deposit/withdraw
✓ Withdraw NO fee compensate (673ms)
✓ Withdraw with GWT compensate (684ms)
✓ Deposit test (337ms)
==5) Matching bonus tests
✓ Matching accrue on withdraw test (5691ms)
✓ Matching expires on timeout (933ms)
✓ Matching on withdraw test. max packs (8069ms)
✓ Matching deposited to frozen (8302ms)
✓ Matching max for 90 days + days added correctly (5333ms)
✓ Days added correctly on expired matching payment (418ms)
==7) Functions tests
✓ Earn limit purchase (721ms)
✓ Buy bPNM purchase limit. BTC rate = 50000 (282ms)
✓ Sell limit purchase (525ms)
✓ Buy 1% marketing (538ms)
✓ Set username (548ms)
✓ Verification tests (777ms)
==8) bPNM buy tests
✓ Buy limit in btcb accrued correctly (216ms)
✓ After prestart mode disabled price is correct (251ms)
✓ Buy bPNM (801ms)
==9) bPNM sell tests
✓ Buy and sell bPNM (811ms)
✓ Sell limit used correctly (7843ms)
==10) PLD test
✓ Liquidity unlocked correctly (447ms)
✓ Liquidity unlocked until zero PLD (331ms)
✓ Correct formula for zero PLD and ZERO usdtFee (375ms)
==11) NFT tests
✓ Rarity set for 10 000 token (396ms)
✓ Token mint voucher accrued, NFT minted, NFT transferred. (880ms)
✓ Mint tokens rounding durin limit pack purchase (358ms)
✓ GWT income for NFT owning (1064ms)
✓ Token URi correct (725ms)

✓ Withdraw with NFT+GWT compensate (1705ms)
✓ NFT discount for limit pack purchase (1929ms)
✓ NFT discount for matching purchase (2549ms)
✓ NFT discount for +1% to marketing level purchase (1994ms)
✓ Reverts tests (252ms)
✓ First 20 tokens pre-minted (479ms)
==12) Marketplace tests
✓ Item add (236ms)
✓ Item can be enabled/disabled. Verify enabled/disabled (355ms)
✓ Purchase item with liquidity compensation (1074ms)
✓ Purchase item NO liquidity compensation (969ms)
✓ Gift item to user (483ms)
✓ Item seller update item (384ms)
✓ Market admin functions (449ms)
==13) Admin funcs
✓ setBuyLimitMultiplier (443ms)
✓ setSellLimitMultiplier (595ms)
✓ setLimitPackPurchaseGwtCompensation (408ms)
✓ setMatchingBonusGwtCost (452ms)
✓ setMatchingBonusExtendPeriod (530ms)
✓ setEarnLimitExtraPerGwt (609ms)
✓ setBuyLimitExtraPerGwt (516ms)
✓ setSellLimitExtraPerGwt (595ms)
✓ setWithdrawBaseFee (533ms)
✓ setBpnmBuyFee (587ms)
✓ setBpnmSellFee (630ms)
✓ setNftMintTokenMaxAmount (227ms)
✓ setNftMintTokenTurnoverRequired (417ms)
✓ setNftDiscountForLimitPackPrice (839ms)
✓ setNftDiscountForMatchingPayment (665ms)
✓ setNftDiscountForAdditionalMarketingPercent (658ms)
✓ setNftDiscountForWithdraw (807ms)
✓ setgwtTransFeeLiquidity (587ms)
✓ changeFeeCollector (413ms)
✓ changeLiquidityCollector (444ms)
✓ changePromoter (407ms)
✓ changeVerificator (408ms)

✓ changeMarketplaceAdministrator (407ms)
✓ setBtcOracle (205ms)
✓ triggerLock (491ms)
✓ Get payment contracts (216ms)
✓ setgwtPerDayForHundredRarity (485ms)
✓ changePromoter_NFT (274ms)
✓ removeAllowedContract_NFT (273ms)
✓ setUnlockPeriod (527ms)
✓ setUnlockPercent (380ms)
✓ changePromoter (369ms)
==14) Test new payment systems
✓ Deposit/withdraw. Payment2 (5028ms)
✓ Deposit/withdraw. Payment3 (424ms)
✓ Activation/Pack buy. Payment2 (1906ms)
✓ Activation/Pack buy. Payment3 (1952ms)

Tests written by Zokyo Security

As a part of our work assisting bPNM in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the bPNM contracts requirements for details about issuance amounts and how the system handles these.

Scope:
bpnmMain.sol
PhenomenalConsultants.sol
phenomenalLiquidityDistributor.sol

```
PhenomenalConsultants
  Deployment
    ✓ Should deploy the PhenomenalConsultants (22083ms)
    ✓ Should set the token name
    ✓ Should set the token name
    ✓ Should set the token symbol
    ✓ Should set the initial owner
    ✓ Should set the promoter
    ✓ Should set the USDT token
    ✓ Should set the BTCB token
    ✓ Should set the GWT token
    ✓ Should set the number of NFTs
  Core functionality
    activate method:
      ✓ Should activate the nwe user (112ms)
      ✓ Should revert error of the verify (72ms)
    replenishPaymentBalance method:
      ✓ Should replenish the user's payment balance (261ms)
    withdrawBalance method:
      ✓ Should withdraw token amount (306ms)
    buyLimitPack method:
      ✓ Should buy limit pack (336ms)
    mintNFT method:
      ✓ Should mint NFT (366ms)
```

```
extendLvlMarketingBonus method:
    ✓ Should extend matching bonus (290ms)
  releaseFrozenFunds method:
    ✓ Should release frozen funds (1800ms)
  buyBpnm method:
    ✓ Should buy bpnm token (788ms)
  sellBpnm method:
    ✓ Should sell bpnm token (855ms)
  buyEarnLimitWithGwt method:
    ✓ Should increase earn limit (495ms)
  buyPurchaseLimit method:
    ✓ Should increase purchase limit (531ms)
  buySellLimit method:
    ✓ Should increase sell limit (789ms)
  extendMatchingBonus method:
    ✓ Should init matching bonus for 30 days (601ms)
    ✓ Should to init matching bonus with discount (199ms)
    ✓ Should to extend matching bonus for 30 days (219ms)
    ✓ Should revert error of max days reached (205ms)
    ✓ Should revert error of not enough balance GWT (137ms)
  toggleLimitPackAutoRenew method:
    ✓ Should toggle limit pack auto renew (459ms)
    ✓ Should revert error if the cost of the package is less than 250 USDT (493ms)
  transfer method:
    ✓ Should not be retold tokens (667ms)
Marketplace methods
  Marketplace method:
    ✓ Should add a new item to the marketplace (370ms)
  triggerMarketItemActive method:
    ✓ Should change the activity status for item (371ms)
  triggerMarketItemVerify method:
    ✓ Should change the item verify requirement (71ms)
  updateMarketItemPrice method:
    ✓ Should change the item price for item (45ms)
  updateMarketItemName method:
    ✓ Should change the name for item (48ms)
```

updateMarketItemClaimLink method:
✓ Should change the link for item (44ms)
giftMarketItemToAddress method:
✓ Should change the item verify requirement (99ms)
Customization methods
setBtcOracle method:
✓ Should revert error if the caller is not the owner (302ms)
✓ Should revert error if the incorrect oracle selector (274ms)
✓ Should change the BTC oracle selector (334ms)
setBuyLimitMultiplier method:
✓ Should revert error if the caller is not the promoter (310ms)
✓ Should revert error if the incorrect oracle range (347ms)
✓ Should change the buy limit multiplier (339ms)
setSellLimitMultiplier method:
✓ Should revert error if the incorrect limit range (346ms)
✓ Should change the sell limit multiplier (365ms)
setLimitPackPurchaseGwtCompensation method:
✓ Should revert error if the incorrect limit range (352ms)
✓ Should change the limit pack purchase gwt compensation (342ms)
setMatchingBonusGwtCost method:
✓ Should revert error if the incorrect limit range (336ms)
✓ Should change the matching bonus gwt cost (432ms)
setMatchingBonusGwtCost method:
✓ Should revert error if the incorrect limit range (342ms)
✓ Should change the matching bonus gwt cost (351ms)
setMatchingBonusExtendPeriod method:
✓ Should revert error if the incorrect limit range (350ms)
✓ Should change the matching bonus extend period (358ms)
setEarnLimitExtraPerGwt method:
✓ Should revert error if the incorrect limit range (352ms)
✓ Should change the earn limit extra per gwt (357ms)
setBuyLimitExtraPerGwt method:
✓ Should revert error if the incorrect limit range (338ms)
✓ Should change the buy limit extra per gwt (340ms)
setSellLimitExtraPerGwt method:
✓ Should revert error if the incorrect limit range (336ms)
✓ Should change the sell limit extra per gwt (324ms)

setWithdrawBaseFee method:
- ✓ Should revert error if the incorrect limit range (355ms)
- ✓ Should change the withdraw base fee (368ms)

setBpnmBuyFee method:
- ✓ Should revert error if the incorrect limit range (306ms)
- ✓ Should change the buy base fee (344ms)

setBpnmSellFee method:
- ✓ Should revert error if the incorrect limit range (318ms)
- ✓ Should change the sell base fee (387ms)

setNftMintTokenMaxAmount method:
- ✓ Should revert error if the incorrect limit range (348ms)
- ✓ Should change the NFT mint max amount (348ms)

setNftMintTokenTurnoverRequired method:
- ✓ Should revert error if the incorrect limit range (341ms)
- ✓ Should change the sell NFT mint turnover required (345ms)

setNftDiscountForLimitPackPrice method:
- ✓ Should revert error if the incorrect limit range (345ms)
- ✓ Should change the NFT discount for limit pack price (321ms)

setNftDiscountForMatchingPayment method:
- ✓ Should revert error if the incorrect limit range (337ms)
- ✓ Should change the NFT discount for limit pack price (371ms)

setNftDiscountForAdditionalMarketingPercent method:
- ✓ Should revert error if the incorrect limit range (340ms)
- ✓ Should change the NFT discount for additional marketing percent (323ms)

setNftDiscountForWithdraw method:
- ✓ Should revert error if the incorrect limit range (341ms)
- ✓ Should change the NFT discount for withdraw (380ms)

setgwtTransFeeLiquidity method:
- ✓ Should revert error if the incorrect limit range (330ms)
- ✓ Should change the gwt trans fee liquidity (392ms)

changeFeeCollector method:
- ✓ Should revert error if the incorrect address (320ms)
- ✓ Should update the fee collector address (291ms)

changeLiquidityCollector method:
- ✓ Should revert error if the incorrect address (321ms)
- ✓ Should update the liquidity collector address (360ms)

changePromoter method:
- ✓ Should revert error if the incorrect address (314ms)

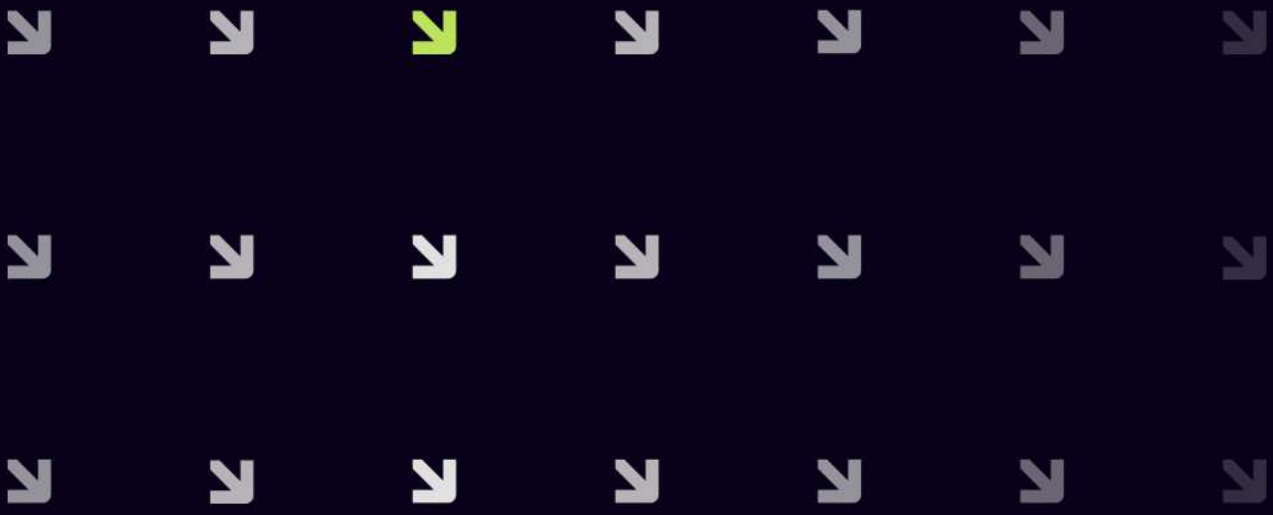
✓ Should update the promoter address (322ms)
changeVerificator method:
✓ Should revert error if the incorrect address (325ms)
✓ Should update the verificator address (322ms)
changeMarketplaceAdministrator method:
✓ Should revert error if the incorrect address (318ms)
✓ Should update the admin address (284ms)
switchPayment method:
✓ Should revert error if the incorrect address (334ms)
✓ Should update the payment contract (394ms)
setUsername method:
✓ Should revert error if the username empty (288ms)
✓ Should revert error if the username pain than 16 symbols (318ms)
✓ Should update the payment contract (354ms)
Scenarios
Scenario 1:
✓ GWT Earnings and Utilization Across Multiple Users (25255ms)
Scenario 2:
✓ Referral System Integrity with Multiple New Users (952ms)
Scenario 3:
✓ NFT Functionality with Multiple Users (699ms)
Scenario 4:
✓ Varied Fee Structures Impact on Transactions (494ms)
Scenario 5:
✓ Payment Methods Variety Test (1265ms)
Scenario 6:
✓ Verifying bpnmPrice Functionality and Decimal Precision

FILE	% STMTS	% BRANCH	% FUNCS
bpnmMain.sol	96.04%	84.67%	100%
PhenomenalConsultants.sol	96.55%	88%	92.31%
phenomenalLiquidityDistributor.sol	100%	88.24%	100%

We are grateful for the opportunity to work with the bPNM team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the bPNM team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.



zokyo