Holyheld: Treasury implementation v0.1 (draft)

Anton Zagorodnikov (antzagos@gmail.com)
Anton Mozgovoy (a@mozgovoy.me)

Abstract

This article describes principles of Holyheld Treasury operation, a component of infrastructure, which purpose is to accumulate value that users can spend on different activities (such as transaction subsidizing, operational fees compensation, etc.) and also serve as an asset pool that backs the value of Holyheld tokens (a portion of this value can be realized by the user through one-time token burn). The goal of the article is to describe Holyheld Treasury's main mechanics, possible action flows, implementation considerations, and interaction with other Holyheld infrastructure components.

1 Treasury concepts

Tokenomics of the Holyheld project [1] is including several features that are coupled with the concept of Holyheld Treasury:

- An incentive to hold for product users: holders of Holyheld Token (hereinafter HH) are accumulating rewards in the form of (currently) USDC token, which cannot be withdrawn directly, but could be used for subsidizing gas cost when performing a transaction for ERC20 token

swap or transfer, covering costs for off-chain operations related to a debit card, etc. This differs from several projects, where the token is the bonus itself and is issued (minted) as rewards without any peg;

- An incentive to hold as a form of cashback on the user capital: a portion of profits generated on user assets is allocated to Treasury. HH token portion could be 'burned', realizing a portion of this pool, decreasing the number of circulating tokens, therefore, allowing for one-time payout rewards for discontinuing usage of the service.

From one point of view, this implies that HH token has a 'book value', that is backed by the value of Treasury and from the other point of view, the market value HH of the token should NOT be seen as a reflection of Holyheld's ability to generate yield (and for bonus portion, the value of its product features).

1.1 Treasury use cases

- Staking and unstaking supported token types into Treasury
  - During deposits users can deposit HH and HH-LP (in a single transaction);
  - During withdrawals users can withdraw HH and HH-LP (in a single transaction), keeping earned bonus amount;
- Holyheld bonus
  - ERC20 Holyheld Bonus token;
  - Claiming any portion of pending rewards as a Holyheld Bonus token;
  - Spending a bonus for supported actions (subsidized transactions and other activities);
- Reward distribution processing – when reward distribution occurs, we count accumulated bonus per staked share and call recalculation (3.1), because it's not dependent on several blocks (and can occur in random periods) and varies in numerical value.

- Claim & Burn action – HH token can be burned from Treasury stake and user balance (or in any proportion of those). This would realize a one-time USDC profit of currently an unspent bonus of user + (4x endowment treasury portion);
- Rebalancing of assets
  - Provide ETH on demand for execution wallets;
  - Rebalance between USDC/ETH/yield-generating assets when thresholds are not met (including deposit/withdraw of USDC into yield-generating pools);

1.2 Treasury Asset Allocation

Treasury assets are divided into two portions: endowment (long-term fund) and bonus pool (assets backing bonuses).

Endowment fund exists because of the following reasons:

- To have a reserved portion that would generate yield on treasury itself;
- To incentivize long-term holding even if bonuses/products are not used;
- To have a Holyheld Bonus token backed by (at least) endowment;
- Everything comes to end eventually, even the stars and probably the universe itself. However, for as long as the Ethereum network will be operational, the treasury would be able to give its assets for HH tokens burn, regardless of other contracts and infrastructure (check how to provide vault safety override to distribute staked assets in case of an emergency event);

The treasury has the following subdivision of assets:

- 50% Endowment threshold. Treasury value may not go below this threshold, even if all bonuses are used. This portion is only claimable through HH token burn. Endowment is usually fully invested in Holypools for the yield generation;
- 50% Bonus pool (portion used for bonus coverage)

– A% of reserves in USDC on the smart contract balance;
– B% of reserves in ETH for gas coverage on the 'execution wallets';
– C% of reserves invested in Holypools for yield generation;

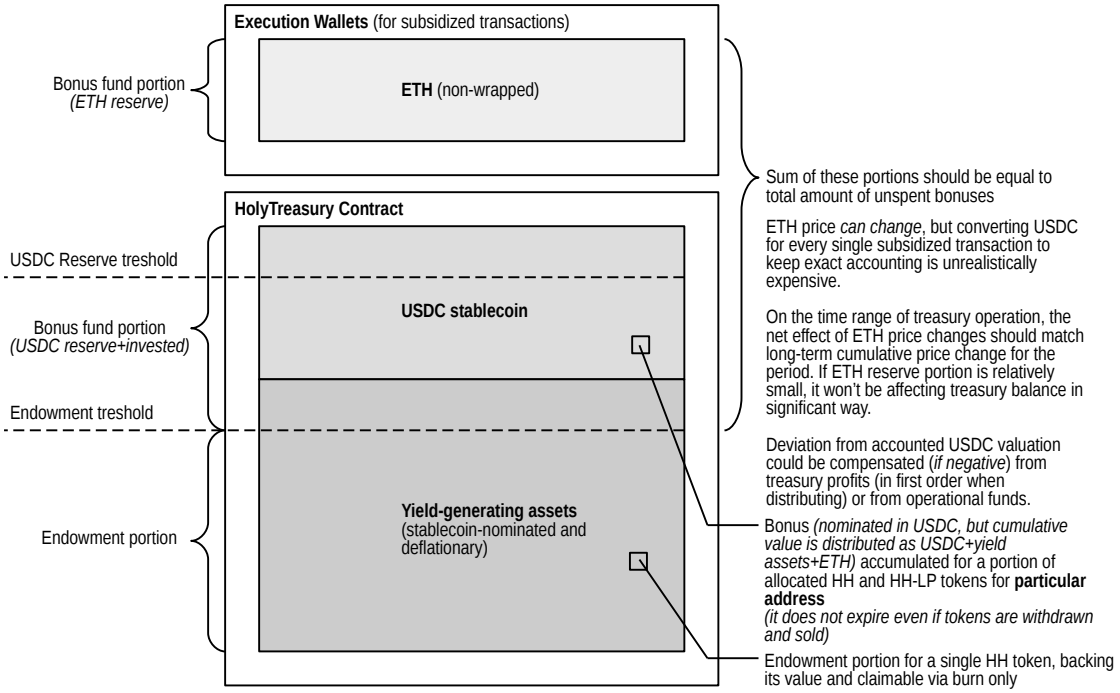

Figure 1: HolyTreasury asset allocation

Endowment percentage proportion to overall treasury size is defined using variable $T_{endowment}$ (`endowment_percent`). This primarily defines *a proportion in which realized profits are accounted in Treasury.*

As Treasury overall assets quantity is dynamic and may significantly differ from defined proportion due to usage of bonuses to cover gas spending and because of asset distribution (across base asset (USDC), ETH (on gas subsidy wallets) and yield-generating assets), a separate variable V_e (`endowment_size`) is used for accounting of endowment asset value. This also would allow changing `endowment_percent` value without introducing accounting errors. Additionally for accounting to avoid bonus balances and

bonus token balances maps iteration for rebalancing checks and other calculations variable V_b (`bonuspool_size`) is defined.

Total amount of Treasury assets could be represented by this equation:

$$V_{total}^{USDC} = V_e + V_b = \sum_{\substack{yield \\ assets}}^{USDC} + USDC_{\substack{contract \\ balance}} + \sum_{\substack{subsidy \\ wallets}}^{USDC} (ETH) \tag{1}$$

From one side, $V_e + V_b$ represent endowment and bonus pool portions (used for accounting), from the other side, assets are allocated to keep a balance between immediate accessibility (ETH) and the ability to generate yield (reserve variables would be monitored and could be changed by balancer service).

The number of wallets used for subsidized transaction execution contains ETH and serves as an immediate gas reserve for executing subsidized transactions. It could be filled from treasury contract by balancer service or when required on immediate demand. Overall reserves for ETH has a settable value (`reserve_eth`);

Treasury has a settable USDC reserve value (`reserve_base_asset`) to be able to provide USDC without conversion overhead when required;

Therefore, treasury seeks to generate yield for itself, while keeping USDC reserve and ETH reserve for cheaper subsidized transaction logistics (and Claim & Burn actions). Treasury itself is staking its portion (endowment plus any other funds in USDC that are above-defined USDC and ETH reserves);

Treasury endowment keeps accounting of its size V_e and should never decrease it except Claim & Burn events.

While using earned bonuses depletes ETH (and eventually, USDC/yield-generating reserves), the percentage portion of bonus pool of Treasury for yield distribution is computed as $T_{bonus} = 100\% - T_{endowment}$ (and getting this portion of profits);

1.3 Arbitrage considerations

As the aim for the platform is to be sustainable, in case of arbitrage opportunities emerge, they should switch balance (HH token price), but not break the tokenomics model:

1. If burn value is *higher* than market value, then it is reasonable to burn the token and buy it on the market; (Burning multiplier would gradually

decrease 'book value' from one side and buying it would increase its market price from the other side, so the new balance should be found in-between).

2. If burn value is *lower* than market value, then token has a premium to book value due to valuation of Holyheld operations as a business, As Treasury performs depositing of itself into yield-generating products, its value should increase even if the volume of other profits is low. The balance between holding due to forward profits participation and valuation of bonuses for in-product use would be found by the market by itself.

1.4 Functional areas breakdown

The contract for Treasury contains several functional areas, that should be separated and orthogonal [5] to reduce security risks and possible side-effects of system behavior.

- Bonus balance accounting: staking/unstaking supported tokens;
- Distribution of yield to bonus and endowment portion, recalculation of valuations and APY metrics;
- Rebalancing of allocation of Treasury assets between a base asset (USDC), ETH hot reserve and yield-generating portions (fulfill USDC and ETH reserves using yield-generating tokens (if their evaluation) is larger than defined amounts; refill ETH reserve from USDC; invest a portion of USDC assets if the amount exceeding reserves is large enough for the operation to be reasonable, etc.)
- Claiming treasury portion through HH token burn;
- Getting ETH for subsidizing a transaction (in case ETH contract needs it);
- ERC20 functions of Holyheld Bonus Token (same contract to avoid allowance costs) and claiming pending bonus as token functions;
- Administration functions (rebalancing thresholds);
- Emergency recover functions (timelocked);

2 Subsidized transactions

By subsidized transactions, we would mean a transaction that is initiated by the user as an action that changes user wallet token balances or user interaction with the Holyheld system. This can include: sending ERC20 tokens, exchanging (swapping) ERC20 tokens, performing deposits and withdrawals in Holyheld yield-generating products, and interaction with Treasury itself (deposits, withdrawals of HH or HH-LP tokens, and HH token burn for claiming a portion of Treasury assets). The term 'subsidized' means that transaction fees (gas fees) are covered for the user.

2.1 Possibility of subsidized transactions

At the time of writing, Ethereum does not provide means to execute transactions with gas provided by another party. But, while having some limitations, such functionality can be implemented. Numerous approaches have emerged, notably the GSN (Gas Station Network) [6].

Holyheld uses similar approach to GSN, which can be concisely described as following steps:

- A user makes a decision to create a transaction. The application determines if such transaction is able to be subsidized technically and if estimated gas expenses cost less than amount user has available as bonus;
- A user signs a message with his private key. The message itself is transparent and clearly states the purpose of the transaction, allowing to match it with the real contract interaction that follows. This serves as a two-way trust: as proof to a system that execution was initiated by the user as the only user has a particular private key, and (the message signatures should be stored persistently) the system can prove to the user that the action was executed because of his signature for a particular timestamped action.
- Signed message and intended transaction content are sent to a backend relay which performs validation checks and initiates a transaction from a wallet holding Eth and providing gas subsidy.

The following transaction types *cannot* be subsidized:

- Raw transfers of Ethereum. There is no mechanism to access ETH on user address (EOA) currently present;
- General third-party smart contract interactions: while some methods in some smart contracts are agnostic to transaction sender, in general, many are relying on `msg.sender`, which would be the address of the backend-controlled wallet. Exceptions could be added manually for possible cases.

The following transaction types *can* be subsidized:

- Sending of ERC20 tokens to any address;
- Swapping of ERC20 tokens using Holyheld swap feature;
- Depositing/withdrawal of appropriate ERC20 tokens into Holyheld yield-generating products or Treasury;
- Burning of HH tokens to receive Treasury portion;

2.2 Execution requirements

Subsidized transactions that require the transfer of ERC20 tokens on the balance of the user wallet could be executed only if sufficient allowance is provided for the Holyheld transfer proxy. Therefore, approval of spending a particular ERC20 token transaction should be executed from the user wallet directly beforehand. The approval amount could be set from minimal value (to execute single action) to larger value (to avoid approval costs for further interactions), depending on security concerns.

Some tokens support the ERC20 Permit feature (EIP-2612) [7]. It allows executing approval from a third-party transaction for an EOA given the signed message (that is verifiable on-chain). In such a case, the approval step can be included in the subsidized transaction itself.

2.3 Transaction creation and verification

Backend should verify (at least) the following information to prevent malicious usage or exploiting the system:

- Match of signature and address provided;

- Signed message contests must be parsed and meet the transaction information provided;
- It should be possible to estimate transaction gas fees;
- User (address) should have enough bonuses to cover transaction gas costs;
- Backend-controlled wallet should have enough ETH to cover transaction gas costs;
- Provided gas price should be reasonably high to avoid stalled transactions;
- Holyheld transfer proxy should have sufficient allowance to spend ERC20 tokens if the transaction requires such interaction or appropriate (EIP-2612 Permit) signed message is provided along with transaction creation request;

2.4 Security and parallelism considerations

As a single address (wallet) on ETH network can only send transactions sequentially, one by one, each increasing `nonce` value, submitting a transaction with a low gas price (that could be done deliberately or unintentionally (gas price quickly rises after submission) could lead to a situation, when a backend-controlled wallet could not send other transactions (they would just be queued), blocking the function for other users.

This can be mitigated partially or completely with following measures:

- Have several backend-controlled wallets with Eth reserves that initiate subsidized transactions. This would not eliminate the issue, but would make the system more robust and stable, allowing to handle a higher rate of subsidized transactions;
- Restricting gas price that user can select to submit subsidized transaction (either completely or use some safety threshold);
- Having backend to cancel transactions that are pending longer than some preset amount of time (overlapping transaction with empty transfer transaction having same `nonce`, using a gas price that is high

enough). Cancellation would require some small amount of gas too and should cost less in gas fees than submitted transaction (except huge gas price spike in a short time, but the likeness of this event probably can be excluded from consideration).

Front-running subsidized transactions for a user is possible, but only if initiated by the user himself (or having provided allowance) and is not economically viable or exploitable.

For example, a user submits a `SEND` order of token T to an address A as a subsidized transaction through Holyheld backend and immediately after that submits a direct transaction to send all token T to address B with the higher gas price. If the latter transaction gets executed before the subsidized one, the subsidized transaction would revert because of an insufficient balance of T. This, however, occurs from the user's consent (or if the user-provided access to token T to an external entity), so we won't be treating this as a security vulnerability.

2.5 Backend execution

Worker thread serving subsidized transactions can follow approximately this logic (very high-level definition), objects manipulation should be thread-safe:

Algorithm 1: Subsidized execution worker thread

```
Result: Pick up and submit transactions from request queue
initialization (connect to Infura, queue, db, retrieve wallets);
while service is active do
    if pending actions on wallets then
        check for timeouts and cancelling settings;
        if stalled transaction then
            cancel pending tx that are stuck for too long (by sending
             empty data Eth transaction)
        end
    end
    while queue has requests AND free wallets available do
        check conditions and that request did not timed out and no
         other requests from this addr pending;
        validate request (including gas price check);
        try other free wallets if balance on current is insufficient;
        if free wallets available, but none with sufficient balance then
            retrieve Eth from Treasury contract;
        end
        execute subsidized transaction;
    end
    if no free wallets available for given time treshold then
        perform alerting activities;
    end
    sleep/idle time quant;
end
```

3 Holyheld Bonus Token

To keep gas expenses minimal, bonus amounts are reflected through values inside of the HolyTreasury contract as a pending bonus and issued as Holyheld Bonus tokens upon staking or unstaking of token types accepted by the Treasury. Bonuses are backed by USDC (for initial implementation, other

stablecoins may be considered in the future), but not directly claimable as USDC tokens. Nonetheless, users can claim bonuses as ERC20 Holyheld Bonus Token. This would make bonuses more robust:

- Bonuses can be transferred between wallets and users;
- Bonus token can be traded, bought, and sold, e.g. via decentralized exchanges;
- There is an incentive to get more bonuses even if their direct use is not intended;
- Various actions, NFT items, and other activities can be implemented requiring a certain number of bonus tokens to be provided (and burned);
- When making subsidized transactions, bonuses accounted in HolyTreasury (used in first-order) and bonus tokens on the wallet balance (used in second-order) can be used together seamlessly;

3.1 Bonus calculation mechanics

To concisely describe the calculation of bonus tokens accounting, a summary of [2, 3]:

Consider a reward program where shares are of a single type and total rewards for participants are allocated at a constant rate per time (block). $shares$ is an overall number of shares, where $share_j$ is the particular number of shares the user has staked.

reward for a particular interval can be described by

$$reward_{t-1,t} = shares_j / shares_{total} * reward_{block} \tag{2}$$

overtime, the $shares_{total}$ change, but keeping the history of such changes could be avoided by calculating accumulated reward per share. Whenever rewards arrive (and per-block rewards use every user deposit or withdrawal call for calculation, as there are no scheduled smart contract calls in Ethereum) a variable that holds accumulated rewards per single share is increased. If there 2x shares for some period, with the same total reward for it, a reward per share increase would be 1/2x. Combined with the mechanics described below, this allows avoiding storing of historical data for overall and per-user shares amounts.

Upon every deposit or withdrawal of a user j, a value of $reward_tally_j$ (rewardDebt) is calculated and stored. All the pending rewards are paid just before that calculation, and essentially, rewardTally sets *zero-point mark*, matching that

$$reward_{pending} = shares_j * accumulated\_reward_{per\_share} - reward_{tally} = 0, \quad (3)$$

where shares_number is *current* shares number after deposit or withdrawal. (like if user was holding current $shares_j$ from the beginning, and just had claimed all his rewards).

Temporal effects of harvesting all pending bonuses when making deposits or withdrawals should not be underestimated: it allows to set 'zero' watermark for a current user standing in regards to accumulated bonus per share. It is possible to recalculate $reward\_tally$ offset for deposits/withdrawals, scaling it to the new number of $shares_j$ while keeping the currently pending bonus value unchanged. Withdrawing all shares while having a pending bonus would result in this value being negative. Thus 'harvesting' of reward during deposit or withdrawal is not only a feature of the approach but serves for simplification, which allows setting 'zero-point' mark with $reward\_tally$, offloading current reward 'chunk' results to another variable (reward balance or minting reward as in [4]), instead of re-scaling it to match currently pending reward.

To summarize: every deposit or withdrawal would first, increase the accumulated reward for the interval between such actions, and second, reset zero marks for the particular user to a new value (realizing pending rewards). This accounts for changing share size from the user side. From the other side, calculating accumulated reward *per share* (acc_reward_per_share) accounts for changing the overall number of shares, taking into consideration staking and unstaking actions of other users.

This approach has a computational complexity of $O(1)$ and storage requirements of 3 variables per user (reward_tally and share_amount, issued bonus or minted tokens amount being the third), making it a viable solution to be implemented in smart-contracts, given their constraints.

This has been done in production numerous times and implementations that use 'multiple staking pools' [4] have separate calculations for every token that can be staked and pools having 'weights' when rewards are distributed (rewards flow is constant, distribution proportion is defined by pool weights), but that does not change calculation mechanics significantly.

The mechanics of rewards (bonus) distribution is closely following approach described in [3], with one additional assumption: $stake_j$ represents stake of a particular user's *LP token type* (e.g. for *HH* and *HH-LP*):

$$stake_{j_{HH}} = \frac{HH_{address}}{HH_{\substack{staked \\ total}}} * w_{HH}; \quad stake_{j_{HH\text{-}LP}} = \frac{HH^{LP}_{address}}{HH^{LP}_{\substack{staked \\ total}}} * w_{HH\text{-}LP}, \tag{4}$$

where w_{HH} is the share weight of single staked HH token and $w_{HH\text{-}LP}$ (values are expected to be $w_{HH} = 1.0$ and $w_{HH-LP} = 2.5$ at the moment of writing).

Fixing the weights and accepted token types from the beginning would allow to have single $reward_t ally$ variable per user or e.g. represent the staked assets issuing a single 'reward-bearing' token (not considering the complexity of converting such token back to some proportion of HH/HH-LP), but would make the solution less flexible.

To be able to change weights without recalculation or have the ability to extend supported token types (and as we should account deposited HH and HH-LP tokens separately in any case), the values of *reward_tally* (`rewardDebt` in [4]) are calculated and stored separately per token type, as do accumulated rewards. This matches the multiple 'staking pools' approaches in [4]. Otherwise (with single reward_tally), decreasing of relative token type weight would decrease pending reward for stakes of that token type, while increasing the pending reward for other token types).

One difference from mentioned approaches is that rewards are used (spent) at arbitrary time when required (regardless of mechanics: through manual claim as ERC20 Holyheld Bonus token, subsidized transaction or other means) and that includes 'pending' rewards (not claimed in the form of token). Pending bonuses are spent in the first order and Holyheld Bonus tokens are used in conjunction if pending bonus amount is not enough. Full spending of pending bonus is equal to harvesting it during deposit or withdrawal, while partial pending bonus usage by increasing reward tally to:

$$reward_{tally} = reward_{tally} + (stake_j * acc_reward_{per\_share}) * \frac{bonus\_to\_spend}{pending_j}, \tag{5}$$

where $bonus\_to\_spend/pending_j$ defines spending fraction or equals 1 when bonus is to be fully spent. The nuance is that pending calculations are

separate for token type, therefore:

$$bonus\_to\_spend_{HH} = bonus\_to\_spend * \frac{pending_{HH}}{pending_{HH} + pending_{HH\text{-}LP}}, \quad (6)$$

where pending reward is defined in (2), if no pending reward available, $reward\_tally$ value is not increased.

As was mentioned, during staking and unstaking assets into Treasury, pending reward amount for the interval between such actions could be offloaded to some 'reward balance' variable to keep calculations simpler. This mechanic could be avoided by rescaling $reward\_tally$ for new share value (7) to match the same pending bonus, saving the cost of changing and storing the value of one more variable:

$$reward^{new}_{tally} = reward^{old}_{tally} + acc\_reward_{per\_share} * (\Delta stake_j), \quad (7)$$

where $\Delta stake_j$ is a change of staked amount for a token type.

In any case, if we need to increase some balance variable, it is semantically (and by gas usage) close to minting a token (Holyheld Bonus token, 3), compacting concepts of pending bonus/bonus balance/bonus tokens to just pending bonus/bonus tokens. The user still could claim a pending bonus as a Holyheld Bonus token at any time (not just during deposits/withdrawals). Architecture-wise, it's more rational (security concerns, calculation, and token logic separation, etc.) to have Holyheld Bonus token harvested whenever possible, than save a tiny amount of gas on deposit/withdraw operations, plus, that would make Holyheld bonus token more liquid.

The other difference is that rewards do not come at a constant rate (e.g. per block), but when yield distribution occurs. Therefore, accumulated bonus per share ($accumulated\_bonus_{per\_share}$) is recalculated not when the user performs deposit or withdraw, but during yield distribution tx. Because such events occur at a random time, this imposes the following behavior: upon yield distribution, bonuses are allocated for the number of staked assets present *at that particular moment*, they are not distributed at a continuous rate.

At the time of writing, it is considered acceptable that if the deposit and withdraw of a certain amount are done between yield distribution events, then bonuses won't be awarded for staking of that particular amount (Fig. 2), but would be awarded for the staked amounts present at t_0 and t_3 (for

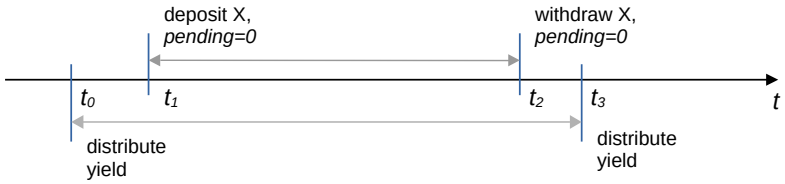

Figure 2: HolyTreasury distribute interval

real use scenario, intervals between deposits and withdrawals like $t_2 - t_1$ are expected to be at least order of magnitude longer than intervals between yield harvests $t_3 - t_0$).

4 Treasury claiming

4.1 Claim-and-burn concept and limitations

As mentioned earlier, Treasury allows performing a 'claim-and-burn' action, meaning a portion (or whole) amount of HH tokens on user's wallet balance along with a portion (or whole) amount of HH tokens staked in Treasury could be burned (destroyed) and for such action, a portion of Treasury *endowment fund* would be paid.

This action would result in less HH tokens circulating supply and a higher level of participation in future profits distribution. Because of such factors, a claim-and-burn operation is planned to be implemented with a multiplier:

$$V_{received} = \frac{HH_{burned}}{HH_{circ.supply}} * V_{treasury}^{USDC} * m_b, \tag{8}$$

where m_b (`burn_multiplier`) is expected to be 4.0 at the time of writing. $HH_{circ.supply}$ is the overall HH token circulating supply, not only the staked HH tokens portion.

This would mean, for example, that burning 50% of HH circulating supply results in getting 200% of endowment, which is unachievable. To prevent such scenarios and involved risks, in single claim-and-burn operation *maximum 5% of the circulating supply could be burned*, still, receiving a significant 20% of Treasury endowment.

4.2 Treasury claim and bonuses

The important difference between Holyheld (HH) token and Holyheld Bonus Token is that Holyheld Bonus Token is minted when the user wants to claim some bonus amount as a token (minting is backed by treasury assets). The amount of HH tokens is fixed and only decreases through the token burn.

If HH token is burned for treasury portion, the portion of bonuses collected by the address performing the burn *would be included up to 100% of treasury portion value received for burning HH token.* For example:

- A user has 100 bonuses (recorded in Treasury contract), 20 HH tokens, and 30 Holyheld Bonus Tokens;
- The user performs 'Claim & Burn' on 10 HH tokens for a reward of treasury value of 50 USDC (this sets the upper limit of bonus conversion to 50 USDC);
- During the Claim & Burn action, 50 of the 100 user bonuses are reclaimed along as USDC;
- After the Claim & Burn action, the user has +100 USDC, 50 bonuses, 10 HH tokens, 30 Holyheld Bonus Tokens;

5 Reference

5.1 Treasury architecture

High-level overview of Treasury components is provided on 3. The only feature that requires the presence of backend is subsidized transactions (this also would be extended to card fees compensation, special events, NFTs, and other activities). The core mechanics of allocating, distributing, and managing bonuses is implemented fully on-chain.

Client-side is represented by components (1) (User-owned Eth Address) and (2) (Application). The application could be a mobile application or other implementations (such as web- or desktop application), the keys for user Ethereum address are never transferred outside the device on which the application is running. Treasury also provides a set of idempotent view functions to the application (balances, APY metrics, etc.).

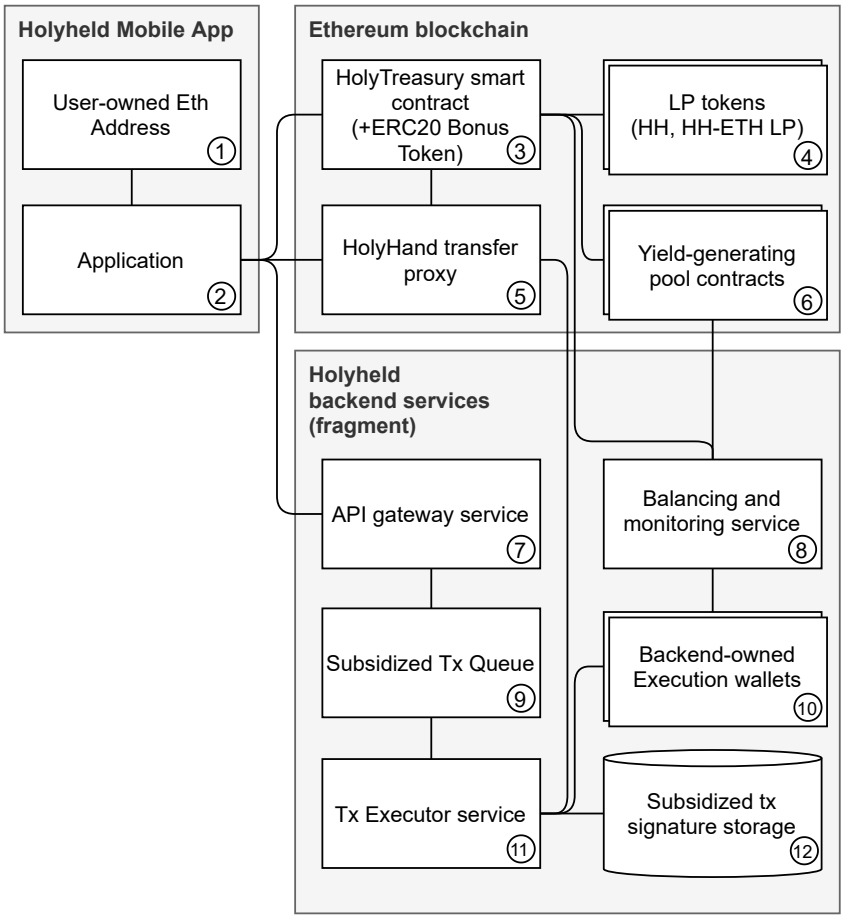

Figure 3: HolyTreasury main components

On Ethereum blockchain, (3) is the main Treasury contract, the application can communicate with it directly for claiming of Holyheld Bonus Token (represented by Treasury contract itself) or transferring it. To save on allowance for depositing tokens supported (4), centralized proxy (5) is used (as for most other token interactions on Holyheld). Treasury can also deposit assets into various yield-generating pool contracts (6) (acting as a user to gain yield on its assets), orchestrated by balancing and monitoring service (8).

For executing subsidized transactions, the application signs messages with

a private key and submits them through API gateway (7), which puts them into queue (9) after initial validation. Messages are being picked up by Tx Executor service (11), which maintains persistent requests storage (12) and a pool of execution wallets (10). Balancing and monitoring service maintains assets allocation and Eth balances on execution wallets to have enough gas as well as providing alerting functions. When yield harvesting occurs on a pool, Treasury acts as a regular user to keep the approach unified, so the balancing service evaluates if asset allocation matches thresholds and if it is viable to execute them.

Backend architecture is fitting into the microservice approach with some of the microservices being stateless and horizontally-scalable, running in a scalable containerized environment, such as Kubernetes, while using cloud KV or database as storage, mainly not for the reason of a high number of requests, but fault tolerance. Running on infrastructure that absorbs several levels of network-related attacks (OSI layer 4 or below [8] [9]) should also be considered.

5.2 Subsidized Tx message formats

This is a non-final description of data that should be included in the signed message. The format is kept concise, but including required action details that were confirmed.

Common message contents	
Data item	Description
`ACTION`	Action to execute (`SEND`, `SWAP`, `DEPOSIT`, `WITHDRAW`)
`TIMESTAMP`	Message confirmation timestamp (when message was signed by user)
`ADDRESS_FROM`	Address from which sending is initiated
`SEND` message contents	
`ADDRESS_TO`	Address to which sending is initiated
`TOKEN`	Address of ERC20 token to be sent
`AMOUNT`	Amount to be sent (`uint`, assuming same number of decimals as specified `TOKEN`)

SWAP message contents	
TOKEN_FROM	Address of ERC20 token to be swapped from (present on wallet)
TOKEN_TO	Address of ERC20 token to be swapped to (to be received after swap)
AMOUNT	Source (from) token amount to be swapped (uint, assuming same number of decimals as specified TOKEN_FROM)
AMOUNT_RCV_MIN	Minimal expected amount of TOKEN_TO to be received, otherwise cancel (revert) the swap (uint, assuming same number of decimals as specified TOKEN_FROM)

DEPOSIT message contents	
TOKEN_FROM	Address of ERC20 token to be deposited (present on wallet)
DESTINATION	Address of smart contract to be deposited to (e.g. HolyPool)
AMOUNT	Source (from) token amount to be swapped (uint, assuming same number of decimals as specified TOKEN_FROM)

WITHDRAW message contents	
DESTINATION	Address of smart contract from which shares should be withdrawn (e.g. HolyPool)
AMOUNT_SHARES or AMOUNT_TOKEN	Amount of shares (or token) to be withdrawn (uint, assuming same number of decimals as specified destination base asset)

5.3 Main variables

endowment_percentage – the percentage of total treasury volume that is counted as an endowment, only claimable through HH token burn;

endowment_size – current valuation in base asset (USDC) of endowment Treasury portion;

bonuspool_size – current valuation in base asset (USDC) of bonus pool Treasury portion;

burn_multiplier – multiplier of Treasury endowment portion to pay for performing token burn along with uspent bonus (limitations described in 4.1);

`reserve_base_asset` – the amount of base asset (USDC) to keep on Treasury contract balance for Claim & Burn function and cheaper conversion to Eth reserves;

`reserve_eth` – the amount of Ethereum to keep on subsidized transaction execution wallets to spend on gas;

5.4 Miscellaneous implementation considerations

- Even if implementation with the arbitrary number of tokens available for deposit (stakeable) tokens would look more generic (more 'mathematically' correct), it would incur more gas expenses due to array iteration (like pools.length in many stakeable contracts [4], etc.);
- We are relying on contract upgradeability to arrange additional stakeable token types (if it would be required);
- To keep contract upgrade-safe, complicated types should be avoided, but for main data mapping (described below) it *should be upgrade-safe* [10];

For a single address (user), several variables would be required in storage (stored as `address => struct[uint256]` mappings):

- `staked_HH` – HH token amount currently deposited to Treasury;
- `staked_HHLP` – HH-ETH LP token amount currently deposited to Treasury;
- `reward_tally_HH` – offset for calculating pending bonuses earned by HH token stake;
- `reward_tally_HHLP` – offset for calculating pending bonuses earned by HH-LP token stake;

`balance`, `allowance` values for ERC20 Holyheld Bonus token are stored with separate mappings as defined in OpenZeppelin Upgradeable Contracts library [11].

References

[1] New 2021, New Holyheld
https://medium.com/holyheld/new-2021-new-holyheld-6980dd576fad

[2] B.Batog et al, Scalable Reward Distribution on the Ethereum Blockchain
http://batog.info/papers/scalable-reward-distribution.pdf

[3] O.Solmaz, Scalable Reward Distribution with Changing Stake Sizes
https://solmaz.io/2019/02/24/scalable-reward-changing/

[4] MasterChef contract (SushiSwap project)
https://github.com/sushiswap/sushiswap/blob/master/contracts/MasterChef.sol

[5] Wikipedia: Orthogonality (programming)
https://en.wikipedia.org/wiki/Orthogonality_(programming)

[6] The Gas Station Network
https://docs.openzeppelin.com/learn/sending-gasless-transactions#gas-station-ne

[7] EIP-2612: permit – 712-signed approvals
https://eips.ethereum.org/EIPS/eip-2612

[8] Open Systems Interconnection model (OSI model)
https://en.wikipedia.org/wiki/OSI_model

[9] Best Practices for DDoS Protection and Mitigation on Google Cloud Platform
https://cloud.google.com/files/GCPDDoSprotection-04122016.pdf

[10] Upgrades with Peace of Mind: "Structs" Edition
https://forum.openzeppelin.com/t/upgrades-with-peace-of-mind-structs-edition/55

[11] OpenZeppelin Contracts Upgradeable
https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable