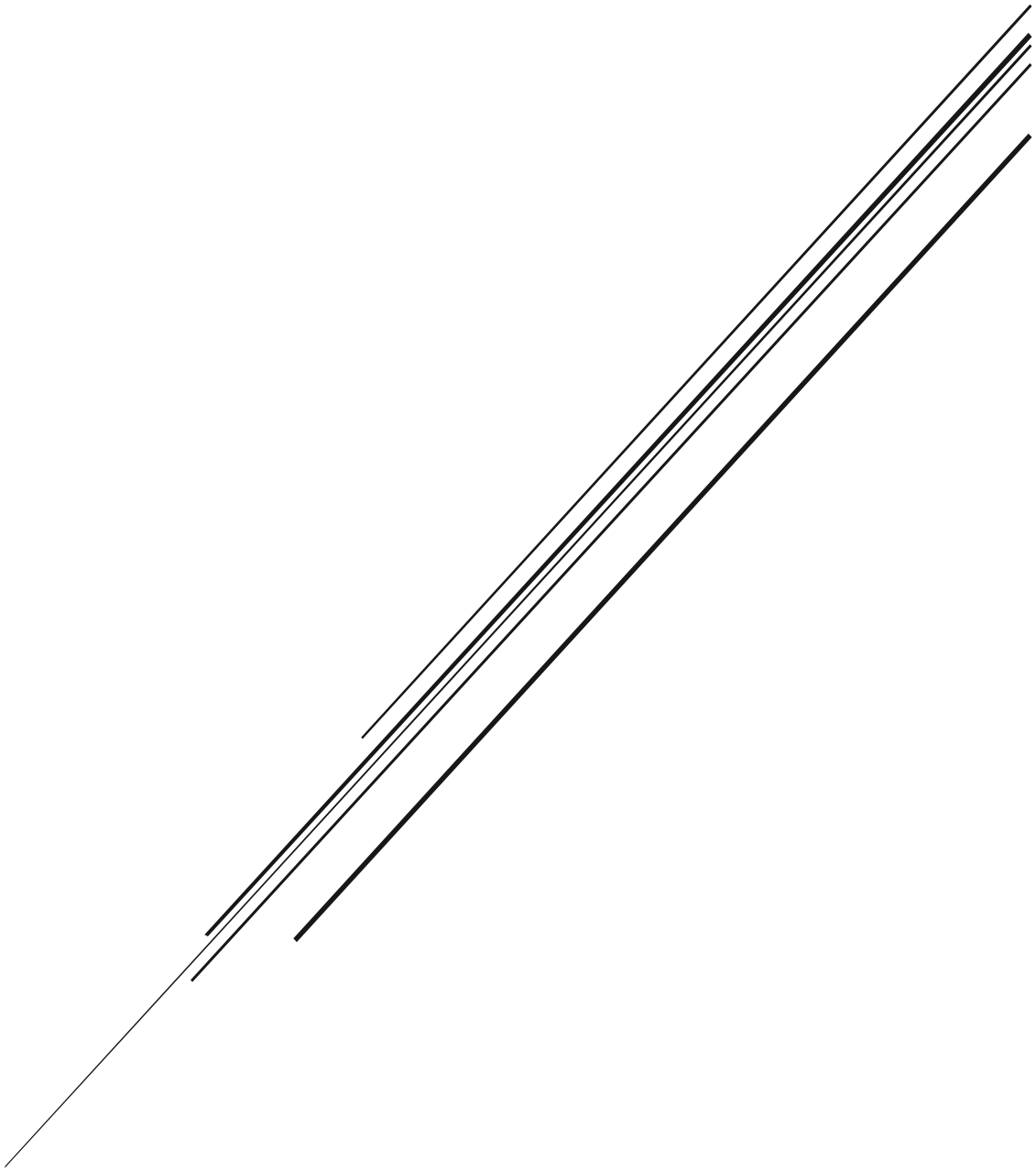
SMART CONTRACT & BLOCKCHAIN DOCUMENTATION

Detailed specifications of the smart contract - HYAX Rewards



HYDRAXIS

Nov 2024

Contents

Use case # 1: Initialize	
Title	Initialize Contract State - initialize(address _hyaxTokenAddress)
Description	This function initializes the smart contract's state variables and inherited contract initializers, sets roles for specific addresses, and validates the HYAX token address.
Actors and interfaces	• Deployer/Contract Owner: Executes the function to initialize the contract. • HYAX Token Contract: Interface used for token-related operations.
Initial status and preconditions	• The address executing the function must have sufficient MATIC to pay the gas fees. • _hyaxTokenAddress must point to a deployed and valid HYAX token contract. • The function must only be called once due to the initializer modifier.
Basic Flow	
Step 1: Initialize state variables of inherited contracts: • Call __AccessControlEnumerable_init(). • Call __ReentrancyGuard_init(). • Call __Pausable_init().	
Step 2: Grant the default admin role to the address executing the function.	
Step 3: Assign specific roles to hardcoded addresses: • Set whiteListerAddress and grant it the WHITELISTER_ROLE. • Set rewardsUpdaterAddress and grant it the REWARDS_UPDATER_ROLE.	
Step 4: Set up the HYAX token address: • Assign _hyaxTokenAddress to the state variable hyaxTokenAddress. • Create an instance of the HYAX token using the address provided.	
Step 5: Initialize state variables for tracking token-related activities: • Set initial values for growth, team, and reward tokens to prevent uninitialized state errors.	
Step 6: Set the initial maximum batch size for updating rewards to 100.	
Step 7: Validate the provided HYAX token address: • Check if the token's symbol matches "HYAX". • Revert the transaction if the validation fails.	
Post Condition	
• Inherited contracts are initialized. • Admin and specific roles (WHITELISTER_ROLE and REWARDS_UPDATER_ROLE) are set up. • State variables for growth, team, and reward tokens are initialized to their default values. • HYAX token address is validated and assigned to the hyaxTokenAddress state variable.	
Alternative flows	

Invalid HYAX Token Address:

- If _hyaxTokenAddress does not point to a contract with the symbol "HYAX", the transaction is reverted with the error message "Hyax token address is not valid".

Reinitialization Attempt:

- If the function is called more than once, the transaction is reverted due to the initializer modifier.

Role Assignment Errors:

- If granting a role fails due to invalid parameters or other issues, the transaction is reverted.

Gas Limit Exceeded:

- If the contract's initialization exceeds the gas limit, the transaction will fail.

Executed by the smart contract implementer (Blockchain Director)

Development Environment -> Polygon Blockchain = Deployed Contract

Use case # 2: Add Wallet to Whitelist	
Title	Add Wallet to Whitelist - addWalletToWhitelist(address _walletAddress,bool _isTeamWallet,uint256 _hyaxHoldingAmountAtWhitelistTime)
Description	Adds a wallet to the whitelist with specific parameters, validating the wallet address, HYAX holding amount, and whether it is a team wallet. Only callable by an admin or whitelister.
Actors and interfaces	• Admin or Whitelister: Can execute this function to whitelist wallets. • Wallet Owner: Indirectly interacts as their wallet gets whitelisted.

Initial status and preconditions	• The function can only be called by an address with the admin or whitelister role.
	• The _walletAddress must:
	  ○ Not be the zero address.
	  ○ Be a valid externally owned account (EOA) and not a smart contract.
	  ○ Not already be whitelisted or blacklisted.
	  ○ Not have been previously added to the whitelist.
	• For team wallets, _hyaxHoldingAmountAtWhitelistTime must be greater than 0 and less than or equal to the total team tokens (TEAM_TOKENS_TOTAL).

	• For non-team wallets, _hyaxHoldingAmountAtWhitelistTime must be 0.
Basic Flow	
Step 1: Validate _walletAddress and its status: • Ensure it is not the zero address. • Confirm it is not a smart contract address. • Check the wallet is not already whitelisted or blacklisted. • Ensure the wallet has not been previously added to the whitelist.	
Step 2: Assign default whitelist properties: • Mark the wallet as whitelisted. • Set it as not blacklisted. • Assign the _isTeamWallet value. • Record the current timestamp as addedToWhitelistTime. • Initialize withdrawal times and rewards-related variables to 0.	

Step 3: Handle team and non-team wallet logic:
• If _isTeamWallet is true:
  ○ Validate _hyaxHoldingAmountAtWhitelistTime is > 0 and ≤ TEAM_TOKENS_TOTAL.
  ○ Set hyaxHoldingAmountAtWhitelistTime and hyaxHoldingAmount to _hyaxHoldingAmountAtWhitelistTime.
• If _isTeamWallet is false:
  ○ Validate _hyaxHoldingAmountAtWhitelistTime is 0.
  ○ Set hyaxHoldingAmountAtWhitelistTime and hyaxHoldingAmount to 0.	

Step 4: Finalize and validate whitelist status: • Confirm the wallet is marked as whitelisted and addedToWhitelistTime is set.	
Step 5: Emit WalletAddedToWhitelist event with relevant details.	
Post Condition	
• The wallet is added to the whitelist with appropriate parameters and initial values. • The WalletAddedToWhitelist event is emitted. • State variables related to the wallet are initialized to default values based on whether it is a team wallet.	
Alternative flows	

Invalid Address:
• If _walletAddress is the zero address or a contract address, the transaction is reverted with appropriate error messages:
  ○ "Address cannot be the zero address"
  ○ "Invalid address length or contract address"
Already Whitelisted or Blacklisted:
• If the wallet is already whitelisted, the transaction reverts with "Wallet is already whitelisted".
• If the wallet is blacklisted, the transaction reverts with "Wallet has been blacklisted".
Team Wallet with Invalid Holding Amount:	

- If _isTeamWallet is true but _hyaxHoldingAmountAtWhitelistTime is ≤ 0 or > TEAM_TOKENS_TOTAL, the transaction reverts with:
    - "Team wallets must be added with a hyax holding amount greater than 0"
    - "Team wallets must be added with a hyax holding amount less than the total team tokens"

Non-Team Wallet with Invalid Holding Amount:

- If _isTeamWallet is false but _hyaxHoldingAmountAtWhitelistTime ≠ 0, the transaction reverts with:
    - "Non team wallets can only be added with holding amount equal to 0"

Final Validation Fails:

- If the wallet is not successfully marked as whitelisted or addedToWhitelistTime is not set, the transaction reverts with "Failed to whitelist the wallet".

Executed by the owner or white lister of the smart contract using their wallet

Backend -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 3: Update Whitelist Status	
Title	Update Wallet Whitelist Status - updateWhitelistStatus(address _walletAddress,bool _newStatus)
Description	Updates the whitelist status of a wallet to a new value. This function ensures the wallet is already in the whitelist and the status differs from the current one. Only callable by an admin or whitelister.
Actors and interfaces	Admin or Whitelister: Can execute the function to update the whitelist status of wallets.

Initial status and preconditions	• The function can only be called by an address with the admin or whitelister role.
	• The _walletAddress must:
	  ○ Have been added to the whitelist (addedToWhitelistTime is not 0).
	  ○ Not already have the same whitelist status as _newStatus.

Basic Flow	
Step 1: Validate _walletAddress and the new status:	

• Check if the current whitelist status of the wallet is different from _newStatus.
  ○ Revert with "Wallet has already been updated to that status" if the status matches.
• Ensure the wallet has previously been added to the whitelist (addedToWhitelistTime ≠ 0).
  ○ Revert with "Wallet has not been added to the whitelist" if not.

Step 2: Update the whitelist status: • Set wallets[_walletAddress].isWhitelisted to _newStatus.
Step 3: Emit the WhitelistStatusUpdated event: • Include the caller's address (msg.sender), the _walletAddress, and the new status (_newStatus).
Post Condition
• The wallet's whitelist status (isWhitelisted) is updated to the specified _newStatus. • The WhitelistStatusUpdated event is emitted to notify of the update.
Alternative flows

Status Already Set:
• If _newStatus matches the wallet's current whitelist status, the transaction reverts with:
  ○ "Wallet has already been updated to that status"
Wallet Not in Whitelist:
• If the wallet has not been previously added to the whitelist (addedToWhitelistTime is 0), the transaction reverts with:
  ○ "Wallet has not been added to the whitelist"

Executed by the owner or white lister of the smart contract using their wallet

Backend -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 4: Update Blacklist Status	
Title	Update Wallet Blacklist Status - updateBlacklistStatus(address _walletAddress,bool _newStatus)
Description	Updates the blacklist status of a wallet to a new value. If blacklisted, the wallet is automatically removed from the whitelist. Only callable by an admin or whitelister.
Actors and interfaces	Admin or Whitelister: Authorized to execute the function to update the blacklist status of wallets.

Initial status and preconditions	• The function can only be called by an address with the admin or whitelister role. • The _walletAddress must have a blacklist status different from _newStatus.
Basic Flow	
Step 1: Validate _walletAddress and the new status: • Check if the current blacklist status of the wallet is different from _newStatus. ○ Revert with "Wallet has already been updated to that status" if the status matches.	
Step 2: Update the blacklist status: • Set wallets[_walletAddress].isBlacklisted to _newStatus.	
Step 3: Adjust whitelist status (if applicable): • If _newStatus is true (wallet is being blacklisted), set wallets[_walletAddress].isWhitelisted to false.	
Step 4: Emit the BlacklistStatusUpdated event: • Include the caller's address (msg.sender), the _walletAddress, and the new blacklist status (_newStatus).	
Post Condition	
• The wallet's blacklist status (isBlacklisted) is updated to the specified _newStatus. • If the wallet is blacklisted (_newStatus == true), its whitelist status (isWhitelisted) is set to false. • The BlacklistStatusUpdated event is emitted to notify of the update.	
Alternative flows	
Status Already Set: • If _newStatus matches the wallet's current blacklist status, the transaction reverts with: ○ "Wallet has already been updated to that status" Wallet Blacklisted: • If _newStatus == true, the wallet is removed from the whitelist (isWhitelisted is set to false).	

Executed by the owner or white lister of the smart contract using their wallet

Backend -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 5: Fund Smart Contract

Title	Fund the Smart Contract with HYAX Tokens - fundSmartContract(FundingType _fundingType, uint256 _amount)
Description	Allows the contract owner to fund the smart contract with HYAX tokens for growth, team, or reward purposes. Ensures that funds comply with predefined limits and the contract's operational status.
Actors and interfaces	• Owner: The administrator of the smart contract, authorized to execute funding operations. • Smart Contract: Receives and processes HYAX token funding.
Initial status and preconditions	• The contract must not be paused. • The function caller must have the DEFAULT_ADMIN_ROLE. • The caller must hold enough HYAX tokens and approve the smart contract for the _amount. • _fundingType must be one of GrowthTokens, TeamTokens, or RewardTokens. • _amount must be greater than 0. • The funding amount for the selected type must not exceed its predefined total limit.

Basic Flow

Step 1: Validate caller and parameters:

- Check that msg.sender is the contract owner.
- Ensure _fundingType is valid (GrowthTokens, TeamTokens, RewardTokens).
- Ensure _amount is greater than 0.

Step 2: Transfer tokens:

- Attempt to transfer _amount HYAX tokens from the owner to the contract.
- If transfer fails, revert with "There was an error on receiving the token funding".

Step 3: Process funding based on _fundingType:

- For GrowthTokens:
  - Validate that growthTokensFunded + _amount <= GROWTH_TOKENS_TOTAL.
  - Update growthTokensFunded and growthTokensInSmartContract.
  - If funding hasn't started, set growthTokensFundingStarted to true, initialize growthTokensStartFundingTime, and growthTokensLastWithdrawalTime.
- For TeamTokens:
  - Validate that teamTokensFunded + _amount <= TEAM_TOKENS_TOTAL.

- Update teamTokensFunded and teamTokensInSmartContract.
- If funding hasn't started, set teamTokensFundingStarted to true and initialize teamTokensStartFundingTime.

- For RewardTokens:
  - Validate that rewardTokensFunded + _amount <= REWARD_TOKENS_TOTAL.
  - Update rewardTokensFunded and rewardTokensInSmartContract.
  - If funding hasn't started, set rewardTokensFundingStarted to true and initialize rewardTokensStartFundingTime.

Step 4: Log the funding type (_fundingType) and amount (_amount).

Post Condition

- The selected token type's funding and smart contract balance are updated with _amount.
- If this is the first funding for the type, its respective "funding started" flag and time variables are initialized.
- The FundingAdded event is emitted.

Alternative flows

Invalid Funding Type:

- If _fundingType is not GrowthTokens, TeamTokens, or RewardTokens, revert with:
  - "Invalid funding type".

Insufficient Token Allowance or Balance:

- If the transfer fails, revert with:
  - "There was an error on receiving the token funding".

Exceeding Funding Limit:

- If _amount exceeds the total limit for the selected token type, revert with:
  - "Amount to fund is greater than the total intended for [token type]".

Paused Contract:

- If the contract is paused, revert due to the isNotPaused modifier.

Reentrancy Attempt:

The function is protected by the nonReentrant modifier to prevent reentrancy attacks.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 7: Withdraw Growth Tokens	
Title	Withdraw Growth Tokens - withdrawGrowthTokens()
Description	Enables the owner to withdraw a fixed yearly amount of growth tokens from the smart contract. Withdrawals are subject to several conditions, such as a one-year interval between withdrawals and remaining tokens available for withdrawal.
Actors and interfaces	• Owner: The only actor authorized to execute this function. • Smart Contract: Validates withdrawal conditions and manages token transfers.
Initial status and preconditions	• The owner receives the withdrawn tokens in their wallet. • The growthTokensWithdrawn and growthTokensInSmartContract variables are updated. • The growthTokensLastWithdrawalTime reflects the time of this withdrawal. • The GrowthTokensWithdrawn event is emitted.
Basic Flow	
Step 1: Validate the caller: • Ensure msg.sender is the contract owner.	
Step 2: Check funding and withdrawal conditions: • Verify that growth tokens funding has started. • Ensure at least one year has passed since funding began (growthTokensStartFundingTime + TOKENS_WITHDRAWAL_PERIOD). • Ensure not all growth tokens have been withdrawn (growthTokensWithdrawn < GROWTH_TOKENS_TOTAL). • Ensure at least one year has passed since the last withdrawal (growthTokensLastWithdrawalTime + TOKENS_WITHDRAWAL_PERIOD).	
Step 3: Determine the withdrawable amount: • Set the initial withdrawableAmount to GROWTH_TOKENS_WITHDRAWAL_PER_YEAR. • Adjust withdrawableAmount if withdrawing the full yearly limit exceeds the remaining balance (GROWTH_TOKENS_TOTAL - growthTokensWithdrawn).	
Step 4: Update state variables: • Increase growthTokensWithdrawn by withdrawableAmount. • Decrease growthTokensInSmartContract by withdrawableAmount. • Update growthTokensLastWithdrawalTime to the current timestamp.	
Step 5: Transfer tokens: • Attempt to transfer withdrawableAmount HYAX tokens to the owner. • If the transfer fails, revert with "Failed to transfer growth tokens".	
Step 6: Emit the GrowthTokensWithdrawn event: • Log the withdrawal, including the caller and the amount withdrawn.	

Post Condition
• The owner receives the withdrawn tokens in their wallet. • The growthTokensWithdrawn and growthTokensInSmartContract variables are updated. • The growthTokensLastWithdrawalTime reflects the time of this withdrawal. • The GrowthTokensWithdrawn event is emitted.
Alternative flows

Unauthorized Caller:
• If msg.sender is not the owner, revert with:
  ○ "Only the owner can withdraw growth tokens".
Funding Not Started:
• If growth tokens funding has not started, revert with:
  ○ "Growth tokens funding has not started yet, no tokens to withdraw".
Insufficient Time Since Last Withdrawal:
• If one year has not passed since the last withdrawal, revert with:
  ○ "Can only withdraw once per year".
All Tokens Withdrawn:
• If all growth tokens have been withdrawn, revert with:
  ○ "All growth tokens have been withdrawn".
Transfer Failure:
• If the token transfer fails, revert with:
  ○ "Failed to transfer growth tokens".

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 8: Withdraw Team Tokens	
Title	Withdraw Team Tokens - withdrawTeamTokens()
Description	Allows team wallets to withdraw their allocated team tokens based on specific conditions, including a 4-year lock period and yearly withdrawal limits. The function checks various conditions to ensure proper execution and prevents reentrancy attacks.
Actors and interfaces	• Team Wallet: The authorized actor to call this function, must meet the necessary conditions (e.g., whitelisted, not blacklisted, has HYAX tokens).

	• Smart Contract: Validates conditions, tracks balances, and facilitates token transfers.
Initial status and preconditions	• The address calling the function must be a whitelisted and non-blacklisted team wallet. • The contract must not be paused. • The team tokens funding must have started. • At least 4 years must have passed since the wallet was added to the whitelist. • There must still be team tokens available for withdrawal (teamTokensWithdrawn < TEAM_TOKENS_TOTAL). • The wallet must have a positive HYAX holding amount. • The wallet must not have exceeded the maximum number of withdrawals allowed per year.
Basic Flow	
Step 1: Validate the caller: • Ensure that the wallet calling the function is a team wallet (wallets[msg.sender].isTeamWallet == true). • Ensure the team tokens funding has started (teamTokensFundingStarted). • Verify the 4-year lock period has passed since the wallet was added to the whitelist (wallets[msg.sender].addedToWhitelistTime + TEAM_TOKENS_LOCKED_PERIOD). • Ensure that not all team tokens have been withdrawn (teamTokensWithdrawn < TEAM_TOKENS_TOTAL). • Ensure the wallet has a positive HYAX holding amount (wallets[msg.sender].hyaxHoldingAmount > 0). • Ensure the wallet has not exceeded the yearly withdrawal limit (wallets[msg.sender].teamTokenWithdrawalTimes < calculateYearForTeamTokens()).	
Step 2: Calculate the withdrawable amount: • Set the initial withdrawable amount as 20% of the wallet's HYAX holding amount at the time of whitelist (wallets[msg.sender].hyaxHoldingAmountAtWhitelistTime / 5). • Check if withdrawing the full amount would exceed the total team tokens. If it does, adjust the withdrawable amount to the remaining balance (TEAM_TOKENS_TOTAL - teamTokensWithdrawn).	
Step 3: Update the contract's internal state: • Subtract the withdrawable amount from the wallet's HYAX holding amount. • Add the withdrawable amount to the total team tokens withdrawn (teamTokensWithdrawn). • Subtract the withdrawable amount from the total team tokens in the smart contract (teamTokensInSmartContract). • Increment the number of times the wallet has made a team token withdrawal (wallets[msg.sender].teamTokenWithdrawalTimes).	
Step 4: Transfer the calculated amount to the wallet:	

- Attempt to transfer the calculated withdrawableAmount of HYAX tokens to the wallet. If the transfer fails, revert with "Failed to transfer team tokens".

Step 5: Emit the TeamTokensWithdrawn event to notify that the team tokens were successfully withdrawn.

Post Condition

- The wallet's HYAX holding amount is decreased by the withdrawn amount.
- The total amount of team tokens withdrawn (teamTokensWithdrawn) is updated.
- The total team tokens in the smart contract (teamTokensInSmartContract) is decreased.
- The wallet's team token withdrawal count is incremented.
- The TeamTokensWithdrawn event is emitted, containing the wallet address and withdrawn amount.

Alternative flows

Unauthorized Wallet:

- If the wallet is not a team wallet, revert with:
  - "Only team wallets can withdraw tokens using this function".

Team Tokens Funding Not Started:

- If team tokens funding hasn't started, revert with:
  - "Team tokens funding has not started yet, no tokens to withdraw".

Too Early for Withdrawal (Before 4 Years):

- If the wallet hasn't waited 4 years after being added to the whitelist, revert with:
  - "Cannot withdraw before 4 years after being added to the whitelist".

All Team Tokens Withdrawn:

- If all team tokens have already been withdrawn, revert with:
  - "All team tokens have been withdrawn".

No HYAX Holding Amount:

- If the wallet has no HYAX holding amount, revert with:
  - "No hyax holding amount to withdraw".

Exceeded Yearly Withdrawal Limit:

- If the wallet has already exceeded the maximum number of withdrawals for the year, revert with:
  - "Can only withdraw team tokens once per year".

Transfer Failure:

- If the transfer of team tokens fails, revert with:
  - "Failed to transfer team tokens".

Executed by a team member with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 9: Calculate Year for Team Tokens

Title	Calculate Year For Team Tokens - calculateYearForTeamTokens()
Description	Calculates the number of years that have elapsed since the team tokens funding started, with a cap at 5 years. This value is used to determine the number of team tokens a wallet can withdraw based on the elapsed time.
Actors and interfaces	Smart Contract Owner: Triggers the function to determine the years since team tokens funding started.
Initial status and preconditions	• Team tokens funding must have started (teamTokensFundingStarted == true). • The contract must be properly initialized with the team tokens funding start time (teamTokensStartFundingTime).

Basic Flow

Step 1: Validate if team tokens funding has started.

- The function checks if teamTokensFundingStarted is true. If not, the transaction is reverted with the error message:
  - "Team tokens funding has not started yet".

Step 2: Calculate the time elapsed since the start of team tokens funding.

- The elapsed time is calculated by subtracting teamTokensStartFundingTime from the current block timestamp (block.timestamp).
- This gives the time in seconds since the funding started.

Step 3: Convert the elapsed time from seconds to years.

- The timeElapsed is divided by 365 days to convert it to full years.

Step 4: Apply the year cap logic.

- If the number of years elapsed is greater than or equal to 8, return 5 (cap at 5 years for withdrawals).
- If the number of years is greater than or equal to 7 but less than 8, return 4.
- If the number of years is greater than or equal to 6 but less than 7, return 3.
- If the number of years is greater than or equal to 5 but less than 6, return 2.
- If the number of years is greater than or equal to 4 but less than 5, return 1.
- If less than 4 years have passed, return 0.

Post Condition

The function does not modify any state variables or balances but returns an integer representing the number of years that have elapsed, capped at 5 years.

Alternative flows

Team Tokens Funding Not Started:

- If the team tokens funding has not started, the transaction is reverted with the message:
    - "Team tokens funding has not started yet".

Negative or Invalid Time Calculation (Unexpected Error):

- In the case of an unexpected error in calculating the time elapsed (e.g., a negative value or incorrect timestamp), the function may revert or return an incorrect value, which is generally handled by the contract's state management and validation logic.

Executed by a user with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Executed internally by the smart contract

Function “withdrawTeamTokens” -> Function “calculateYearForTeamTokens” -> Consultation with internal smart contract value -> Return of value to the smart contract

Use case # 10: Update Rewards Batch	
Title	Update Rewards Batch - updateRewardsBatch(address[] calldata _walletAddresses,uint256[] calldata _hyaxRewards)
Description	Updates the rewards for a batch of wallet addresses. It iterates through the provided list of wallets and attempts to update the rewards for each. If an error occurs during the update of a specific wallet, a RewardUpdateFailed event is emitted.
Actors and interfaces	• Admin or Rewards Updater: The user executing the function. This role is required to trigger the function. • Wallets: The wallet addresses whose rewards are being updated.
Initial status and preconditions	• The address executing the function must have the onlyAdminOrRewardsUpdater role (i.e., admin or designated rewards updater). • Reward tokens funding must have started (rewardTokensFundingStarted == true).

	• The wallet address list (_walletAddresses) cannot be empty and must not exceed the defined batch size (maximumBatchSizeForUpdateRewards). • The array lengths of _walletAddresses and _hyaxRewards must match.
Basic Flow	
Step 1: Validate the function preconditions. • Ensure that reward tokens funding has started (rewardTokensFundingStarted == true). If not, the transaction is reverted with the message: ○ "Reward tokens funding has not started yet, no tokens to update". • Ensure that the array _walletAddresses is not empty, and does not exceed the defined batch size (maximumBatchSizeForUpdateRewards). • Ensure that the length of _walletAddresses matches the length of _hyaxRewards.	
Step 2: Iterate through the wallet addresses provided in the _walletAddresses array. • For each wallet address at index i, attempt to update its reward using the corresponding value from _hyaxRewards[i].	
Step 3: Handle errors during the reward update. • If an error occurs during the update of a particular wallet's rewards (via updateRewardsSingle), the function catches the error and emits a RewardUpdateFailed event, including the error message and the wallet address.	
Step 4: Emit an event signaling the completion of the batch update. • After iterating through all wallets, emit the RewardUpdateBatchSent event, containing the list of wallet addresses and their corresponding reward amounts.	
Post Condition	
• The rewards for each wallet address in the batch are updated (if no errors occurred). • If any errors occurred, those specific updates are logged via RewardUpdateFailed. • A RewardUpdateBatchSent event is emitted to notify about the batch update.	
Alternative flows	
Precondition Validation Failure: • If the preconditions fail (such as if the funding has not started, array lengths mismatch, or batch size limit is exceeded), the transaction is reverted with an appropriate error message. Error in Updating Specific Wallet Rewards: • If an error occurs when updating a specific wallet's rewards (e.g., during the call to updateRewardsSingle), the function catches the error and emits the RewardUpdateFailed event with the specific error message for that wallet. Successful Batch Update:	

- If all reward updates are successful, the RewardUpdateBatchSent event is emitted, signaling the completion of the batch update process.

Executed by the owner or rewards updater of the smart contract using their wallet

Backend -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 11: Update Rewards for a Single Wallet	
Title	Update Rewards for a Single Wallet - updateRewardsSingle(address _walletAddress, uint256 _hyaxRewards)
Description	Updates the rewards for a single wallet address after validating conditions such as wallet status, reward limits, and funding availability. If the update is successful, the wallet's reward balances are updated, and a success event is emitted. If an error occurs, a failure event is triggered.
Actors and interfaces	• Admin or Rewards Updater: The user executing the function, who must have the onlyAdminOrRewardsUpdater role. • Wallet: The wallet address whose rewards are being updated.
Initial status and preconditions	• The address executing the function must have the onlyAdminOrRewardsUpdater role. • Reward tokens funding must have started (rewardTokensFundingStarted == true). • The wallet to be updated must be whitelisted (wallets[_walletAddress].isWhitelisted == true) and not blacklisted (wallets[_walletAddress].isBlacklisted == false). • The wallet must not have been updated too soon after the last reward update (block.timestamp >= wallets[_walletAddress].lastRewardsUpdateTime + MIN_INTERVAL_FOR_UPDATE_REWARDS). • The reward amount (_hyaxRewards) must not exceed the weekly reward limit (REWARD_TOKENS_PER_WEEK). • The contract must have sufficient reward tokens available (_hyaxRewards <= rewardTokensInSmartContract).

	• The total distributed rewards must not exceed the total reward limit (rewardTokensDistributed + _hyaxRewards <= REWARD_TOKENS_TOTAL).
Basic Flow	

Step 1: Validate the preconditions.
• Ensure that reward tokens funding has started (rewardTokensFundingStarted == true). If not, the transaction is reverted with the message:
  ○ "Reward tokens funding has not started yet, no tokens to update".
• Check if the wallet is whitelisted (wallets[_walletAddress].isWhitelisted == true). If not, the transaction is reverted with the message:
  ○ "Wallet is not whitelisted".
• Ensure that the wallet is not blacklisted (wallets[_walletAddress].isBlacklisted == false). If the wallet is blacklisted, the transaction is reverted with the message:
  ○ "Wallet has been blacklisted".
• Check if the rewards update interval has passed (block.timestamp >= wallets[_walletAddress].lastRewardsUpdateTime + MIN_INTERVAL_FOR_UPDATE_REWARDS). If it hasn't, the transaction is reverted with the message:
  ○ "Too soon to update rewards for this wallet".	

Step 2: Additional reward validation.
• Ensure that the reward amount (_hyaxRewards) does not exceed the weekly reward limit (_hyaxRewards <= REWARD_TOKENS_PER_WEEK). If it does, the transaction is reverted with the message:
  ○ "A single wallet cannot have rewards higher than the weekly limit".
• Ensure that the contract has sufficient reward tokens to distribute (_hyaxRewards <= rewardTokensInSmartContract). If not, the transaction is reverted with the message:
  ○ "Insufficient reward tokens to distribute as rewards".
• Ensure that the total distributed rewards will not exceed the total reward cap (rewardTokensDistributed + _hyaxRewards <= REWARD_TOKENS_TOTAL). If exceeded, the transaction is reverted with the message:
  ○ "All the reward tokens have been already distributed".	

Step 3: Update wallet and contract states.
• Update the wallet's last rewards update timestamp to the current block time (wallets[_walletAddress].lastRewardsUpdateTime = block.timestamp).
• Increment the total rewards distributed (rewardTokensDistributed += _hyaxRewards).
• Add the reward amount to the wallet's total and current reward balances:
  ○ wallets[_walletAddress].totalHyaxRewardsAmount += _hyaxRewards
  ○ wallets[_walletAddress].currentRewardsAmount += _hyaxRewards.	

Step 4: Emit the success event.	

- Emit the RewardUpdateSuccess event, signaling the successful update of rewards for the wallet, including the sender's address, the wallet address, and the reward amount.

Post Condition

- The wallet's reward balances are updated: totalHyaxRewardsAmount and currentRewardsAmount are increased by the new rewards amount.
- The contract's total distributed rewards (rewardTokensDistributed) are increased by the new rewards amount.
- The wallet's lastRewardsUpdateTime is updated to the current block time.
- The RewardUpdateSuccess event is emitted, indicating the successful update.

Alternative flows

Invalid Precondition:

- If any of the preconditions are not met (e.g., funding not started, wallet not whitelisted, etc.), the transaction is reverted with an appropriate error message.

Exceeding Reward Limits:

- If the reward amount exceeds the weekly limit or available tokens in the contract, the transaction is reverted with the respective error message:
  - "A single wallet cannot have rewards higher than the weekly limit".
  - "Insufficient reward tokens to distribute as rewards".
  - "All the reward tokens have been already distributed".

Error in Updating Wallet Rewards:

- If any unexpected error occurs during the reward update process, the transaction is reverted, and no updates are made.

Executed by the owner or rewards updater of the smart contract using their wallet

Backend -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 12: Withdraw Reward Tokens

Title	Withdraw Reward Tokens – withdrawRewardTokens()
Description	This function allows a whitelisted and non-blacklisted user to withdraw their accumulated reward tokens. It validates various conditions, including the availability of reward tokens, and ensures the contract is not paused before

	proceeding with the transfer. An event is emitted upon a successful withdrawal.
Actors and interfaces	User (msg.sender): The address initiating the withdrawal of reward tokens.
Initial status and preconditions	• The address executing the function must be whitelisted (isWhitelisted(msg.sender)) and not blacklisted (isNotBlacklisted(msg.sender)). • The contract must not be paused (isNotPaused). • Reward token funding must have started (rewardTokensFundingStarted == true). • There must be reward tokens available for withdrawal (rewardTokensWithdrawn < REWARD_TOKENS_TOTAL). • The user must have a positive reward balance (wallets[msg.sender].currentRewardsAmount > 0). • There must be sufficient reward tokens in the contract to complete the withdrawal (withdrawableAmount <= rewardTokensInSmartContract).
Basic Flow	

Step 1: Validate the preconditions.
• Ensure that reward tokens funding has started (rewardTokensFundingStarted == true). If not, the transaction is reverted with the error:
  ○ "Reward tokens funding has not started yet, no tokens to withdraw".
• Ensure that the total amount of reward tokens withdrawn has not reached the limit (rewardTokensWithdrawn < REWARD_TOKENS_TOTAL). If all reward tokens have been withdrawn, the transaction is reverted with the error:
  ○ "All reward tokens have been withdrawn".
• Retrieve the user's available rewards to withdraw from their wallet state (uint256 withdrawableAmount = wallets[msg.sender].currentRewardsAmount).
• Ensure that the user has rewards to withdraw (withdrawableAmount > 0). If not, the transaction is reverted with the error:
  ○ "No rewards available to withdraw".
• Ensure that there are enough reward tokens available in the contract (withdrawableAmount <= rewardTokensInSmartContract). If not, the transaction is reverted with the error:
  ○ "Insufficient reward tokens in the contract to withdraw".	

Step 2: Reset the user's reward state. • Set the user's currentRewardsAmount to 0, indicating the rewards have been withdrawn (wallets[msg.sender].currentRewardsAmount = 0).	
Step 3: Update the user's total rewards withdrawn. • Increase the user's rewardsWithdrawn by the withdrawn amount (wallets[msg.sender].rewardsWithdrawn += withdrawableAmount).	
Step 4: Update the last withdrawal time for the user.	

- Set the user's lastRewardsWithdrawalTime to the current block timestamp (wallets[msg.sender].lastRewardsWithdrawalTime = block.timestamp).

Step 5: Update the smart contract's reward token state.

- Decrease the contract's rewardTokensInSmartContract by the withdrawn amount (rewardTokensInSmartContract -= withdrawableAmount).
- Increase the total reward tokens withdrawn (rewardTokensWithdrawn += withdrawableAmount).

Step 6: Transfer the reward tokens.

- Call the transfer function of the hyaxToken contract to send the withdrawn tokens to the user's address (bool transferSuccess = hyaxToken.transfer(msg.sender, withdrawableAmount)).
- If the transfer fails, the transaction is reverted with the error:
  - "Failed to transfer reward tokens".

Step 7: Emit the success event.

- Emit the RewardTokensWithdrawn event with the user's address and the withdrawn amount (emit RewardTokensWithdrawn(msg.sender, withdrawableAmount)).

Post Condition

- The user's currentRewardsAmount is set to 0.
- The user's rewardsWithdrawn is updated by the withdrawn amount.
- The user's lastRewardsWithdrawalTime is updated to the current block timestamp.
- The contract's rewardTokensInSmartContract is decreased by the withdrawn amount.
- The total withdrawn amount (rewardTokensWithdrawn) is updated.
- The reward tokens are successfully transferred to the user's wallet.
- The RewardTokensWithdrawn event is emitted, confirming the withdrawal.

Alternative flows

Preconditions Not Met:

- If any of the following conditions fail, the transaction is reverted with the corresponding error message:
  - "Reward tokens funding has not started yet, no tokens to withdraw".
  - "All reward tokens have been withdrawn".
  - "No rewards available to withdraw".
  - "Insufficient reward tokens in the contract to withdraw".

Transfer Failure:

- If the transfer function fails (e.g., due to an error in transferring the reward tokens), the transaction is reverted with the error message:
  - "Failed to transfer reward tokens".

Executed by a whitelisted user with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 13: Withdraw Tokens to Burn	
Title	Withdraw Tokens to Burn - withdrawTokensToBurn(FundingType _fundingType, uint256 _amount)
Description	This function allows the default admin to withdraw tokens to be burned, with validation for the token type and amount. The withdrawal is only possible if the respective funding has started and there are enough tokens in the contract. An event is emitted upon a successful withdrawal.
Actors and interfaces	Default Admin (msg.sender): The address that can initiate the withdrawal of tokens to be burned.
Initial status and preconditions	• The address executing the function must have the DEFAULT_ADMIN_ROLE. • The contract must not be paused (isNotPaused). • The _fundingType must be one of the following valid types: GrowthTokens, TeamTokens, or RewardTokens. • The _amount must be greater than 0. • The respective funding type must have started (i.e., growthTokensFundingStarted, teamTokensFundingStarted, or rewardTokensFundingStarted). • The contract must have enough tokens in the relevant category to fulfill the withdrawal (i.e., growthTokensInSmartContract, teamTokensInSmartContract, or rewardTokensInSmartContract).
Basic Flow	
Step 1: Validate that the caller has the default admin role. • Check that msg.sender has the DEFAULT_ADMIN_ROLE (i.e., the function is only accessible by the admin).	

Step 2: Validate the token withdrawal conditions.
• Ensure that _fundingType is valid (either GrowthTokens, TeamTokens, or RewardTokens). If invalid, the transaction is reverted with the error:
  ○ "Invalid funding type".
• Ensure that _amount is greater than 0. If not, the transaction is reverted with the error:
  ○ "Amount must be greater than 0".	

Step 3: Validate the specific conditions for the chosen funding type.

- For GrowthTokens:
  - Ensure that growth token funding has started (growthTokensFundingStarted == true). If not, the transaction is reverted with the error:
    - "Growth tokens funding has not started yet, no tokens to withdraw".
  - Ensure there are enough growth tokens in the contract to withdraw (_amount <= growthTokensInSmartContract). If not, the transaction is reverted with the error:
    - "Insufficient growth tokens in the contract to withdraw".
  - Decrease the amount of growth tokens in the smart contract by the withdrawn amount (growthTokensInSmartContract -= _amount).
- For TeamTokens:
  - Ensure that team token funding has started (teamTokensFundingStarted == true). If not, the transaction is reverted with the error:
    - "Team tokens funding has not started yet, no tokens to withdraw".
  - Ensure there are enough team tokens in the contract to withdraw (_amount <= teamTokensInSmartContract). If not, the transaction is reverted with the error:
    - "Insufficient team tokens in the contract to withdraw".
  - Decrease the amount of team tokens in the smart contract by the withdrawn amount (teamTokensInSmartContract -= _amount).
- For RewardTokens:
  - Ensure that reward token funding has started (rewardTokensFundingStarted == true). If not, the transaction is reverted with the error:
    - "Reward tokens funding has not started yet, no tokens to withdraw".
  - Ensure there are enough reward tokens in the contract to withdraw (_amount <= rewardTokensInSmartContract). If not, the transaction is reverted with the error:
    - "Insufficient reward tokens in the contract to withdraw".
  - Decrease the amount of reward tokens in the smart contract by the withdrawn amount (rewardTokensInSmartContract -= _amount).

Step 4: Transfer the specified amount to the owner.

- Call the transfer function of the hyaxToken contract to send the specified amount of tokens to the owner's address (require(hyaxToken.transfer(owner(), _amount), "Failed to withdraw tokens")).
- If the transfer fails, the transaction is reverted with the error:
  - "Failed to withdraw tokens".

Step 5: Emit the withdrawal event.

- Emit the TokensToBurnWithdrawn event to notify that tokens have been withdrawn for burning (emit TokensToBurnWithdrawn(_fundingType, _amount)).

Post Condition

- The relevant token balance in the smart contract (growth, team, or reward) is decreased by the withdrawn amount.
- The TokensToBurnWithdrawn event is emitted, confirming the withdrawal of tokens for burning.

Alternative flows

Invalid Funding Type:

- If the _fundingType is not one of the allowed types (GrowthTokens, TeamTokens, or RewardTokens), the transaction is reverted with the error:
  - "Invalid funding type".

Zero or Negative Amount:

- If the _amount is 0 or less, the transaction is reverted with the error:
  - "Amount must be greater than 0".

Insufficient Tokens in the Contract:

- If the contract does not have enough tokens of the specified type (growth, team, or reward) to fulfill the withdrawal, the transaction is reverted with the appropriate error message:
  - "Insufficient growth tokens in the contract to withdraw".
  - "Insufficient team tokens in the contract to withdraw".
  - "Insufficient reward tokens in the contract to withdraw".

Non-admin User:

- If the caller does not have the DEFAULT_ADMIN_ROLE, the transaction is reverted with the error:
  - "Only the owner can withdraw tokens".

Transfer Failure:

- If the transfer fails (e.g., due to insufficient allowance or a failure in the transfer function), the transaction is reverted with the error:
  - "Failed to withdraw tokens".

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 14: Update Team Member Wallet Address

Title	Update Team Member Wallet Address - updateTeamMemberWallet(address _oldTeamMemberWalletAddress,address _newTeamMemberWalletAddress)
Description	This function updates the wallet address of a team member. It transfers the wallet data from the old team member wallet to the new wallet, ensuring the old wallet is no longer marked as a team wallet and the new wallet is marked as such.
Actors and interfaces	• Contract Owner: The address executing the function (must have the DEFAULT_ADMIN_ROLE). • Team Member Wallet: The old and new team member wallet addresses.
Initial status and preconditions	• The address executing the function must have the DEFAULT_ADMIN_ROLE and be the contract owner. • The contract must have started the team tokens funding (i.e., teamTokensFundingStarted must be true). • The old wallet address must already be a valid team wallet (marked as isTeamWallet). • The old wallet address must be whitelisted and not blacklisted. • The new wallet address must not be zero, must not be the same as the old address, and must not already be whitelisted or marked as a team wallet. • The new wallet address must also not be blacklisted.
Basic Flow	
Step 1: The contract owner calls the updateTeamMemberWallet function with the old and new wallet addresses as parameters.	
Step 2: The function checks if the caller is the contract owner by validating the DEFAULT_ADMIN_ROLE and ensures the team tokens funding has started (teamTokensFundingStarted is true).	
Step 3: The function validates the conditions for the old and new wallet addresses: • The old wallet must be a team wallet and whitelisted. • The old wallet must not be blacklisted. • The new wallet address must not be the zero address and must not match the old wallet address. • The new wallet must not already be whitelisted or marked as a team wallet and must not be blacklisted.	
Step 4: The wallet data from the old wallet is transferred to the new wallet, including the hyaxHoldingAmountAtWhitelistTime, hyaxHoldingAmount, addedToWhitelistTime, and teamTokenWithdrawalTimes.	

Step 5: The old wallet's status is updated to indicate it is no longer whitelisted, no longer a team wallet, and is now blacklisted.

Step 6: The new wallet's status is updated to indicate it is whitelisted, marked as a team wallet, and not blacklisted.

Step 7: The function verifies the correctness of the data transfer by checking the new wallet's whitelist status and ensuring the old wallet's values have been cleared.

Step 8: The TeamMemberWalletUpdated event is emitted, notifying the network of the update, including the new wallet address and updated team token holdings.

Post Condition

- The old team member wallet is no longer whitelisted, no longer marked as a team wallet, and has its data cleared.
- The new team member wallet is whitelisted, marked as a team wallet, and contains the transferred wallet data (e.g., team token holdings).
- The contract emits a TeamMemberWalletUpdated event to notify of the wallet change.

Alternative flows

- Invalid Caller: If the caller is not the contract owner or does not have the DEFAULT_ADMIN_ROLE, the function will revert with the message "Only the owner can update the team member wallet."
- Team Tokens Funding Not Started: If the team tokens funding has not started (teamTokensFundingStarted is false), the function will revert with the message "Team tokens funding has not started yet, no tokens to recover."
- Old Wallet Invalid: If the old wallet address is not a team wallet, not whitelisted, or blacklisted, the function will revert with appropriate messages:
    - "Old wallet address is not a team wallet"
    - "Old team member wallet address is not whitelisted"
    - "Old team member wallet address is blacklisted"
- New Wallet Invalid: If the new wallet address is invalid (e.g., zero address, already whitelisted, already a team wallet, or blacklisted), the function will revert with the corresponding error message:
    - "New team member wallet address cannot be the zero address"
    - "New team member wallet address cannot be the same as the old team member wallet address"
    - "New team member wallet address is already whitelisted"
    - "New team member wallet address is already a team wallet"
    - "New team member wallet address is blacklisted"
- Data Integrity Failure: If any validation step fails during the process (e.g., incorrect data transfer or wallet status update), the function will revert with the message "Failed to update the team member wallet."

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 15: Update White Lister Address	
Title	Update White Lister Address - updateWhiteListerAddress(address _whiteListerAddress)
Description	This function allows the admin to update the white lister address. It validates the new address, revokes the role from the old white lister, grants the role to the new address, and emits an event to notify the update.
Actors and interfaces	Admin (only role authorized to call this function) White lister (previous and new white lister address)
Initial status and preconditions	• The address executing this function must have the DEFAULT_ADMIN_ROLE. • The _whiteListerAddress provided must not be the zero address. • The contract must already have a whiteListerAddress set (the previous white lister).
Basic Flow	
Step 1: The admin calls the function updateWhiteListerAddress with the new white lister's address as an argument.	
Step 2: The function checks that the new white lister address is not the zero address.	
Step 3: The previous white lister's role is revoked using revokeRole.	
Step 4: The new white lister is granted the WHITELISTER_ROLE using grantRole.	
Step 5: The whiteListerAddress variable is updated to the new address.	
Step 6: The function emits the WhiteListerAddressUpdated event, notifying the update with the new white lister address.	
Post Condition	
• The previous white lister no longer has the WHITELISTER_ROLE. • The new white lister address is now the active address with the WHITELISTER_ROLE. • The whiteListerAddress global variable is updated to the new address.	
Alternative flows	
• If the _whiteListerAddress is the zero address, the transaction reverts with the message: "White lister address cannot be the zero address." • If the address executing the function is not the admin (does not have DEFAULT_ADMIN_ROLE), the transaction reverts with an error due to the onlyRole(DEFAULT_ADMIN_ROLE) modifier.	

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 16: Update Rewards Updater Address	
Title	Update Rewards Updater Address - updateRewardsUpdaterAddress(address _rewardsUpdaterAddress)
Description	This function allows the admin to update the rewards updater address by validating the new address, revoking the role from the old updater, granting the role to the new address, and emitting an event to notify the change.
Actors and interfaces	• Admin: The entity authorized to call this function (must have DEFAULT_ADMIN_ROLE). • Rewards Updater: The previous and new rewards updater addresses.
Initial status and preconditions	• The address executing this function must hold the DEFAULT_ADMIN_ROLE. • The _rewardsUpdaterAddress parameter must not be the zero address. • The contract must already have a rewardsUpdaterAddress set (representing the previous updater).
Basic Flow	
Step 1: The admin calls the updateRewardsUpdaterAddress function, passing the new rewards updater address as a parameter.	
Step 2: The function validates that the new address is not the zero address. If the address is invalid, the transaction reverts with an error message.	
Step 3: The contract revokes the REWARDS_UPDATER_ROLE from the current rewardsUpdaterAddress.	
Step 4: The contract assigns the REWARDS_UPDATER_ROLE to the _rewardsUpdaterAddress.	
Step 5: The rewardsUpdaterAddress variable is updated to the new address.	
Step 6: The function emits the RewardsUpdaterAddressUpdated event, including the new rewards updater address in the event payload.	
Post Condition	
• The previous rewards updater address no longer holds the REWARDS_UPDATER_ROLE.	

• The new rewards updater address is now the active holder of the REWARDS_UPDATER_ROLE. • The rewardsUpdaterAddress variable is updated to the new address. • An event is emitted to notify about the address change.
Alternative flows
• If _rewardsUpdaterAddress is the zero address, the transaction is reverted with the error message: "Rewards updater address cannot be the zero address". • If the caller does not have the DEFAULT_ADMIN_ROLE, the transaction is reverted due to the onlyRole(DEFAULT_ADMIN_ROLE) modifier.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 17: Update HYAX Token Contract Address	
Title	Update HYAX Token Contract Address - updateHyaxTokenAddress(address _hyaxTokenAddress)
Description	Allows the admin to update the address of the HYAX token contract after verifying the new address is valid, owned by the caller, and corresponds to a legitimate HYAX token.
Actors and interfaces	• Admin: The authorized entity to call this function (must have DEFAULT_ADMIN_ROLE). • HYAX Token Contract: The new token contract being set.
Initial status and preconditions	• The address executing the function must hold the DEFAULT_ADMIN_ROLE. • The caller must be the owner of the contract (msg.sender == owner()). • The _hyaxTokenAddress parameter must not be the zero address. • The token at _hyaxTokenAddress must have the symbol "HYAX" to be considered valid.
Basic Flow	
Step 1: The admin calls updateHyaxTokenAddress, providing the _hyaxTokenAddress as a parameter.	
Step 2: The function checks that the caller is the owner of the contract. If not, the transaction reverts with the error message: "Only the owner can update the hyax token address".	

Step 3: The function validates that _hyaxTokenAddress is not the zero address. If it is, the transaction reverts with the error message: "Hyax token address cannot be the zero address".
Step 4: The function initializes an instance of the HYAX token interface (IHyaxToken) using the provided address.
Step 5: The function verifies that the token at the provided address has the symbol "HYAX". If it does not, the transaction reverts with the error message: "Token address must be a valid HYAX token address".
Step 6: The contract updates the hyaxTokenAddress to the new address.
Step 7: The contract updates the hyaxToken instance to reference the new token contract.
Step 8: The function emits a HyaxTokenAddressUpdated event, logging the new token contract address.
Post Condition
• The hyaxTokenAddress variable is updated to the new address. • The hyaxToken variable is updated to reference the new HYAX token instance. • The contract can now interact with the new HYAX token. • An event is emitted to notify about the update.
Alternative flows
• If the caller is not the owner of the contract, the transaction is reverted with the error message: "Only the owner can update the hyax token address". • If _hyaxTokenAddress is the zero address, the transaction is reverted with the error message: "Hyax token address cannot be the zero address". • If the token at _hyaxTokenAddress does not have the symbol "HYAX", the transaction is reverted with the error message: "Token address must be a valid HYAX token address".

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 18: Update Maximum Batch Size for Updating Rewards

Title	Update Maximum Batch Size for Updating Rewards - updateMaximumBatchSizeForUpdateRewards(uint8 _maximumBatchSizeForUpdateRewards)

Description	Updates the maximum number of rewards that can be processed in a batch. The admin must ensure the new batch size is within the allowed range (1 to 100).
Actors and interfaces	Admin: Authorized to update the maximum batch size (must have DEFAULT_ADMIN_ROLE).
Initial status and preconditions	• The address executing the function must hold the DEFAULT_ADMIN_ROLE. • The _maximumBatchSizeForUpdateRewards parameter must be greater than 0 and less than or equal to 100.
Basic Flow	
Step 1: The admin calls updateMaximumBatchSizeForUpdateRewards, providing the _maximumBatchSizeForUpdateRewards parameter**.**	
Step 2: The function validates that _maximumBatchSizeForUpdateRewards is greater than 0. If not, the transaction reverts with the error message: "Maximum batch size cannot be 0".	
Step 3: The function validates that _maximumBatchSizeForUpdateRewards does not exceed 100. If it does, the transaction reverts with the error message: "Maximum batch size cannot be greater than 100".	
Step 4: The function updates the maximumBatchSizeForUpdateRewards variable with the provided value.	
Step 5: The function emits the MaximumBatchSizeForUpdateRewardsUpdated event to notify about the update.	
Post Condition	
• The maximumBatchSizeForUpdateRewards variable is updated with the new value. • An event is emitted to log the updated batch size.	
Alternative flows	
• If the caller does not have the DEFAULT_ADMIN_ROLE, the transaction is reverted. • If _maximumBatchSizeForUpdateRewards is 0, the transaction is reverted with the error message: "Maximum batch size cannot be 0". • If _maximumBatchSizeForUpdateRewards exceeds 100, the transaction is reverted with the error message: "Maximum batch size cannot be greater than 100".	

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 19: Get Contract Owner Address	
Title	Get Contract Owner Address – owner()
Description	Retrieves the address of the contract owner by returning the first account assigned the DEFAULT_ADMIN_ROLE.
Actors and interfaces	Anyone: This is a public view function accessible by any user or smart contract.
Initial status and preconditions	The smart contract must have at least one address assigned the DEFAULT_ADMIN_ROLE. The caller must have sufficient gas for the read operation (though minimal).
Basic Flow	
Step 1: The caller invokes the owner function.	
Step 2: The function retrieves the first address associated with the DEFAULT_ADMIN_ROLE using getRoleMember(DEFAULT_ADMIN_ROLE, 0).	
Step 3: The function returns the retrieved address to the caller.	
Post Condition	
• The function returns the address of the first account with the DEFAULT_ADMIN_ROLE. • No state changes occur in the smart contract.	
Alternative flows	
If no address is assigned the DEFAULT_ADMIN_ROLE, the function may revert internally when calling getRoleMember.	

Executed by a user with a crypto wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 20: Pause	
Title	Pause Contract Functionality - pause()
Description	Pauses all functionalities of the smart contract by invoking the internal _pause function. Can only be called by an admin.
Actors and interfaces	Smart Contract Admin: The user with the DEFAULT_ADMIN_ROLE is allowed to execute this function.
Initial status and preconditions	• The caller must have the DEFAULT_ADMIN_ROLE. • The contract must not already be in a paused state. • The address executing the function must have sufficient gas to perform the transaction.
Basic Flow	

Step 1: The caller executes the pause function.
Step 2: The function checks if the caller has the DEFAULT_ADMIN_ROLE.
Step 3: If the caller is authorized, the _pause internal function is invoked.
Step 4: The _pause function sets the contract's state to paused, preventing further use of certain functionalities until resumed.
Post Condition
• The contract enters a paused state, disabling all functions marked with the whenNotPaused modifier. • The contract emits a Paused event, signaling the pause action to external systems.
Alternative flows
• Unauthorized Caller: If a caller without the DEFAULT_ADMIN_ROLE attempts to execute the function, the transaction is reverted with an error. • Already Paused State: If the contract is already paused, the _pause function may revert depending on its implementation.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 21: Unpause	
Title	Unpause Contract Functionality - unpause()
Description	Resumes normal functionality of the contract by calling the internal _unpause function. Can only be executed by an admin.
Actors and interfaces	Smart Contract Admin: The user with the DEFAULT_ADMIN_ROLE who can execute this function.
Initial status and preconditions	• The caller must have the DEFAULT_ADMIN_ROLE. • The contract must currently be in a paused state. • The address executing the function must have sufficient gas to perform the transaction.
Basic Flow	
Step 1: The caller executes the unpause function.	
Step 2: The function verifies that the caller has the DEFAULT_ADMIN_ROLE.	
Step 3: If the caller is authorized, the _unpause internal function is invoked.	
Step 4: The _unpause function updates the contract's state to unpaused, allowing all functionalities marked with the whenNotPaused modifier to resume.	
Post Condition	

• The contract exits the paused state, and all functionalities restricted by the whenNotPaused modifier become accessible. • A Unpaused event is emitted to signal that the contract is unpaused.
Alternative flows
• Unauthorized Caller: If a caller without the DEFAULT_ADMIN_ROLE attempts to execute the function, the transaction is reverted with an error. • Already Unpaused State: If the contract is not in a paused state, the _unpause function may revert or take no action, depending on its implementation.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 22: Transfer Contract Ownership	
Title	Transfer Contract Ownership - transferOwnership(address newOwner)
Description	Transfers the ownership of the contract to a specified new address by granting the DEFAULT_ADMIN_ROLE to the new owner and revoking it from the current owner.
Actors and interfaces	• Smart Contract Admin: The current owner (with the DEFAULT_ADMIN_ROLE) who initiates the ownership transfer. • New Owner: The address that will become the new owner of the contract.
Initial status and preconditions	• The caller must have the DEFAULT_ADMIN_ROLE. • The newOwner address must not be the zero address (address(0)) or the contract address itself. • The address executing the function must have sufficient gas to perform the transaction.
Basic Flow	
Step 1: The current owner calls the transferOwnership function with the newOwner address as an argument.	
Step 2: The function verifies that the caller is the current owner using the onlyRole(DEFAULT_ADMIN_ROLE) modifier and the owner() check.	
Step 3: The function validates that the newOwner address is not the zero address (address(0)).	
Step 4: The function ensures that the newOwner address is not the same as the contract address.	

Step 5: The DEFAULT_ADMIN_ROLE is granted to the newOwner address using grantRole.
Step 6: The DEFAULT_ADMIN_ROLE is revoked from the current owner using revokeRole.
Post Condition
• The new owner is assigned the DEFAULT_ADMIN_ROLE. • The current owner no longer has the DEFAULT_ADMIN_ROLE. • The contract's ownership is effectively transferred to the newOwner.
Alternative flows
Unauthorized Caller: If the caller does not have the DEFAULT_ADMIN_ROLE, the transaction is reverted. Invalid newOwner Address: • If newOwner is address(0), the transaction is reverted with the error "Ownable: new owner is the zero address". • If newOwner is the contract's address, the transaction is reverted with the error "Ownable: new owner cannot be the same contract address". Failure in Granting or Revoking Role: If granting or revoking the role fails due to external reasons, the function execution is halted.

Executed by the owner of the smart contract using his wallet

Website -> Connection to wallet with ethers.js -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 23: (Proxy Admin) Renounce Ownership

Title	Renounce Ownership - renounceOwnership()
Description	This function allows the current owner to renounce ownership of the contract, transferring ownership to the zero address (address(0)). After renouncing, no functions protected by the onlyOwner modifier can be called, effectively leaving the contract without an owner.
Actors and interfaces	• Contract Owner: The only actor who can call this function to renounce ownership. • The function interacts with the internal ownership management mechanism, specifically transferring ownership to address(0).
Initial status and preconditions	• The contract must have an owner, and the address executing the function must be the current owner. • The address executing the function must have sufficient gas to complete the transaction.

	• The function is protected by the onlyOwner modifier, which ensures that only the current owner can call it.
Basic Flow	
Step 1: The current owner calls the renounceOwnership() function.	
Step 2: The contract checks if the caller is the current owner using the onlyOwner modifier.	
Step 3: The contract executes the _transferOwnership(address(0)) function, transferring ownership to the zero address.	
Step 4: Ownership is effectively renounced, and the contract is left without an owner.	
Step 5: The function completes successfully, and the contract no longer has an owner.	
Post Condition	
• The contract no longer has an owner, as ownership has been transferred to address(0). • Any functions or actions restricted to the owner (using the onlyOwner modifier) are now inaccessible. • The state of the contract is such that it is effectively ownerless and cannot be managed by a specific address.	
Alternative flows	
Caller is Not the Owner: • If the caller is not the contract owner, the onlyOwner modifier will prevent the transaction from proceeding, and it will revert with the error message *"Ownable: caller is not the owner"*. Invalid Contract State: • If there is any issue preventing the _transferOwnership() function from executing properly (e.g., an internal failure), the transaction will revert and not renounce ownership.	

Executed by an owner with a crypto wallet

Website -> Connection to wallet with Mult signature Wallet -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 24: (Proxy Admin) Transfer Ownership	
Title	Transfer Ownership - transferOwnership(address newOwner)
Description	This function transfers ownership of the contract to a new address (newOwner).

	It updates the _owner state variable and emits an OwnershipTransferred event to signal the change in ownership. The function is protected by the onlyOwner modifier, which ensures that only the current owner can call it.
Actors and interfaces	The function interacts with the internal _owner state variable, which holds the address of the current contract owner.
Initial status and preconditions	• The contract must have an existing owner (a valid address is stored in the _owner variable). • The function does not require any specific parameters to be met beyond the valid newOwner address being passed to it by the calling function.
Basic Flow	
Step 1: The newOwner address is passed as a parameter to the function.	
Step 2: The function stores the current _owner in the variable oldOwner.	
Step 3: The _owner state variable is updated to the newOwner address.	
Step 4: The function emits an OwnershipTransferred event, passing the oldOwner and newOwner addresses.	
Step 5: The ownership is successfully transferred, and the function execution completes.	
Post Condition	
• The _owner state variable is updated to reflect the newOwner address. • The contract's ownership has been transferred to the new address. • An OwnershipTransferred event is emitted, notifying the change of ownership.	
Alternative flows	
Invalid Address for newOwner: • If the function is called with an invalid or inappropriate address for newOwner, the transaction may fail (although the function itself doesn't enforce validation checks). If validation logic is implemented externally, such as in transferOwnership, the transaction may revert before _transferOwnership is invoked. No Change in Ownership: • If the newOwner address is the same as the current _owner, the function will still emit the OwnershipTransferred event, but the ownership state will remain unchanged. If additional checks are added (e.g., ensuring the newOwner is not the same as the current owner), the transaction might be reverted.	

Executed by an owner with a crypto wallet

Website -> Connection to wallet with Mult signature Wallet -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Use case # 25: (Proxy Admin) Upgrade and Call	
Title	Upgrade and Call - upgradeAndCall(ITransparentUpgradeableProxy proxy, address implementation, bytes memory data)
Description	The function upgrades the specified proxy contract to a new implementation and optionally calls a function on the upgraded implementation. It requires the contract to be the admin of the proxy. If no data is provided, no ETH is sent with the transaction.
Actors and interfaces	• Contract Owner: The owner is the only actor who can call this function as it is restricted by the onlyOwner modifier. • ITransparentUpgradeableProxy: This is the interface for the proxy contract that will be upgraded. • Implementation Contract: The contract that will replace the current implementation in the proxy.
Initial status and preconditions	• The contract executing this function must be the **admin** of the proxy contract (i.e., it must have the required permissions to upgrade the proxy). • The proxy contract (proxy) must be deployed and must be an instance of the ITransparentUpgradeableProxy interface. • The address provided for the implementation must be a valid contract address. • If no data (data) is passed, the transaction must not include any value (msg.value must be zero). If data is provided, msg.value can be any value, depending on the needs of the function called on the new implementation.
Basic Flow	
Step 1: The contract owner calls upgradeAndCall, passing the proxy contract address (proxy), the new implementation contract address (implementation), and the data (data) to execute on the upgraded implementation.	
Step 2: The contract checks that the sender is the contract owner (via the onlyOwner modifier).	
Step 3: The function calls the upgradeToAndCall method on the proxy contract, passing the implementation address and the data.	
Step 4: If data is not empty, the function call specified in data is executed on the new implementation contract. If data is empty, no function call is executed, but the contract is upgraded.	
Step 5: If msg.value is provided, it is forwarded to the proxy.upgradeToAndCall function.	
Step 6: The proxy contract is now upgraded to the new implementation, and any additional function call (if provided) is executed.	

Post Condition

- The proxy contract is upgraded to the new implementation (implementation).
- If data was provided, the specified function on the new implementation is executed.
- The proxy is now using the upgraded logic provided by the new implementation contract.
- The state of the contract (in terms of the proxy's logic) has changed, and any funds (msg.value) sent with the transaction (if applicable) are handled by the proxy contract.

Alternative flows

Proxy Admin Validation:

- If the contract calling upgradeAndCall is not the admin of the proxy, the transaction is reverted with an error indicating insufficient permissions.

Empty data with Non-Zero msg.value:

- If the data is empty but msg.value is non-zero, the transaction may revert if it violates the condition requiring msg.value to be zero when data is empty.

Invalid Implementation Address:

- If the implementation address is invalid or the contract at the implementation address does not implement the expected logic, the transaction may fail, and the upgrade operation will not occur.

Failure in Function Call on Implementation:

- If the data is provided and the function call specified in data fails (e.g., due to invalid parameters or a reverted transaction), the entire transaction will be reverted.

Executed by an owner with a crypto wallet

Website -> Connection to wallet with Mult signature Wallet -> Invocation of function execution using the wallet -> Execution on Polygon Blockchain -> Response to backend with ethers.js -> Show information on the front

Smart Contract implementation process

1. The process starts using a code editor such as **Visual Studio code** in which the smart contract will be programmed using the **Solidity** programming language.

2. Subsequently, the functional, security and gas tests of the smart contract are generated using the **JavaScript programming language,** (Security testing is not a substitute for security audits.)

3. **Yarn** is used to manage the dependencies required by the smart contract, while **Hardhat** is used as a development environment to compile, debug and deploy the smart contract in the testnet.

4. It is possible that to make the development of the smart contract more efficient, Scaffold-ETH2 will be used. At the same time, **Gitlab** is used to perform code version control.

5. With the aim of realistically testing the smart contract, it will be deployed on the **Amoy testnet network** , which allows you to interact with the smart contract from a blockchain explorer such as **Polygon Scan .**

6. Finally, to deploy the smart contract in production, the **Polygon Blockchain** and **hardhat** will be used to carry out the deployment.

7. Once the smart contract is uploaded to the blockchain, it will be possible to interact with it using the **ethers.js** library and a provider like **metamask** , using **ethers.js** and the **Alchemy API** or using a crypto asset wallet from a blockchain explorer as in the case of **Polygon Scan** .

References

- Smart contract programming language: Solidity version +0.8.0
  Documentation: https://docs.soliditylang.org/en/v0.8.21/

- Smart contract testing programming language: Javascript latest version
  Documentation: https://developer.mozilla.org/en-US/docs/Web/JavaScript

- Code editor: Visual studio code
  Documentation: https://code.visualstudio.com/docs

- Development environment to compile, debug and deploy smart contracts : Hardhat
  Documentation: https://hardhat.org/docs

- Package manager to manage dependencies in the project: Yarn
  Documentation: https://yarnpkg.com/getting-started

- Toolkit to make smart contract development more efficient: Scaffold-ETH2
  Documentation: https://scaffold-eth-2-docs.vercel.app/

- Version control systems for source code management in software development: Github / Gitlab
  Documentation: https://docs.gitlab.com/

- Blockchain explorer to access and interact with smart contracts : Polygon scan
  Documentation: https://docs.Polygonscan.com/

- Blockchain for testing replica of the Polygon blockchain: Amoy testnet
  Documentation: https://forum.polygon.technology/t/introducing-the-amoy-testnet-for-polygon-pos/13388

- Blockchain to deploy smart contracts in production: Polygon blockchain
  Documentation: https://www.coinbase.com/es/learn/crypto-basics/what-is-Polygon

    - Crypto token addresses over polygon testnet (amoy)
        - POL Native token. Must be obtained from Faucet https://faucet.polygon.technology/
        - https://www.alchemy.com/faucets/polygon-amoy