Owners Pool

Contract address:
0x46a122B50E7fAEC5b0cFc526EC9494D9a68A5011

The OwnersPool contract manages a shared pool of funds for a group of owners with predefined shares. It supports secure allocation, withdrawal of funds, and balance updates, leveraging role-based access control to ensure operational security. The contract is built on Solidity ^0.8.0 and uses OpenZeppelin's AccessControl library for managing permissions.This contract is reserved exclusively for owner members, this pool accumulates a percentage of tokens from each package purchase.
The contract ensures security through several measures. Role-based access control restricts critical operations, such as setting owner details, to operators with the OPERATOR_ROLE, and withdrawals are limited to owners with allocated balances. Solidity ^0.8.0 provides built-in overflow protection, enhancing reliability in calculations. Additionally, the contract securely interacts with the USDT token using the IERC20 interface, and ownership details are immutable once set, preventing unauthorized tampering.
This design supports practical use cases like revenue sharing among stakeholders based on predefined shares and collaborative ventures where funds are proportionally pooled. Key advantages include transparency, allowing owners to view balances and shares through getOwnerInfo, and event logging for all changes. The _update function automates fund allocation, reducing errors, and the implementation leverages OpenZeppelin's AccessControl for robust security.

Libraries and Interfaces

OpenZeppelin AccessControl:
Used to implement role-based access control, ensuring that only authorized parties can perform critical operations like setting owners. Role management by assigning:
- DEFAULT_ADMIN_ROLE: Assigned to the administrator, allowing them to manage other roles.
- OPERATOR_ROLE: Assigned to operators who can set owner addresses and shares.

IERC20_USDT Interface:
Defines key ERC20 functions (transferFrom, transfer, balanceOf) for interaction with the Tether (USDT) token contract.

Contract Functions

Readable functions

Function	Description
DEFAULT_ADMIN_ROLE	A constant that defines the default administrative role in the smart contract. Used to manage access control.
OPERATOR_ROLE	A constant that defines the role for operators who may have specific permissions within the contract.
allocatedBalance	Returns the portion of contract balance that is allocated to the owners.
getOwnerInfo(_owner address)	Returns the share and balance information for a specified owner.
getOwners	Returns the list of all owner addresses.
getRoleAdmin(role bytes32)	A function to retrieve the administrator responsible for managing a specific role within the contract.
hasRole(role bytes32, account address)	A function that checks whether a specific account holds a particular role in the contract.
supportsInterface(interfaceId bytes4)	Checks if the contract implements a specific interface (useful for compatibility and standards like ERC165).
tetherToken	Returns the address of a linked Tether (USDT) token.

Executable Functions

Function	Description
_update()	Updates each owner’s balance based on their share of the unallocated USDT in the contract. Allocates funds proportionally to each owner. Updates the allocatedBalance to include the newly allocated funds.
grantRole(role bytes32, account address)	Assigns a specified role to an account, giving it specific permissions or access within the smart contract.
renounceRole(role bytes32, callerConfirmation address)	Allows an account to voluntarily relinquish a role it holds, removing its associated permissions.
revokeRole(role bytes32, account address)	Used by an administrator to remove a specific role from an account, effectively revoking its permissions.
setOwners(_ownerAddresses address[], _shares uint256[])	Allows the operator to set owner addresses and their respective shares. Ensures the shares total exactly 100 percent. Prevents re-setting of owners once finalized (ownersIsSet). Emits the OwnerAdded event for each owner.
withdrawal()	Allows an owner to withdraw their allocated balance from the pool. Requires that the caller has a positive balance. Transfers the USDT balance to the owner and reduces the allocatedBalance. Emits the Withdrawal event.

Codes

Function	**Code**
DEFAULT_ADMIN_ROLE	Public variable
OPERATOR_ROLE	Public variable
allocatedBalance	Public variable
getOwnerInfo(_owner address)	function getOwnerInfo(address _owner) external view returns (Owner memory) { return ownerInfo[_owner]; }
getOwners	function getOwners() external view returns (address[] memory) { return owners; }
getRoleAdmin(role bytes32)	function getRoleAdmin(bytes32 role) public view virtual returns (bytes32) { return _roles[role].adminRole; }
hasRole(role bytes32, account address)	function hasRole(bytes32 role, address account) public view virtual returns (bool) { return _roles[role].hasRole[account]; }
supportsInterface(interfaceId bytes4)	function supportsInterface(bytes4 interfaceId) public view virtual returns (bool) { return interfaceId == type(IERC165).interfaceId; }
tetherToken()	Public variable
_update()	function _update() public { uint256 currentBalance = tetherToken.balanceOf(address(this)) - allocatedBalance; if (currentBalance > 0) { uint256 eachShareAmount = currentBalance / 10; for (uint256 i = 0; i < owners.length; i++) { address owner = owners[i]; Owner storage ownerData = ownerInfo[owner]; uint256 ownerAmount = (eachShareAmount * ownerData.share); ownerData.balance += ownerAmount; } } allocatedBalance += currentBalance; }
grantRole(role bytes32, account address)	function grantRole(bytes32 role, address account) public virtual onlyRole(getRoleAdmin(role)) { _grantRole(role, account); }

Function	**Code**
renounceRole(role bytes32, callerConfirmation address)	`function renounceRole(bytes32 role, address callerConfirmation) public virtual { if (callerConfirmation != _msgSender()) { revert AccessControlBadConfirmation(); } _revokeRole(role, callerConfirmation); }`
revokeRole(role bytes32, account address)	`function revokeRole(bytes32 role, address account) public virtual onlyRole(getRoleAdmin(role)) { _revokeRole(role, account); }`
setOwners(_ownerAddresses address[], _shares uint256[])	`function setOwners(address[] memory _ownerAddresses, uint256[] memory _shares) external onlyRole(OPERATOR_ROLE) { require(!ownersIsSet, "Owners aleady set."); require(_ownerAddresses.length == _shares.length, "Must provide same length"); uint256 _totalShare; for (uint i = 0; i < _shares.length; i++) { _totalShare += _shares[i]; } require(_totalShare == 10, "Invalid shares"); ownersIsSet= true; for (uint i = 0; i < _ownerAddresses.length; i++) { require(_ownerAddresses[i] != address(0), "Invalid address provided"); owners.push(_ownerAddresses[i]); ownerInfo[_ownerAddresses[i]] = Owner({ share: _shares[i], balance: 0 }); emit OwnerAdded(_ownerAddresses[i], _shares[i]); } }`
withdrawal()	`function withdrawal() external updatePool { Owner storage ownerData = ownerInfo[msg.sender]; uint256 payout = ownerData.balance; require(payout > 0, "No funds to claim"); ownerData.balance = 0; tetherToken.transfer(msg.sender, payout); allocatedBalance -= payout; emit Withdrawal(address(this), msg.sender, payout); }`