SAFE KEEP CONTRACT ANALYSIS

Indept analysis of the Safe Keep VaultFacet.sol smart contract and it's. dependencies.

BY: AMARACHI UGWU

NB: VaultFacet.sol is a part(facet) of the Safe Keep project built with Diamond Standard. SafeKeep is a social recovery platform that allows users to get back access to their vault even when they lose admin access to it.

Solidity Version: pragma solidity 0.8.4;

This version allows for custom errors to be defined inside and outside of contracts (including interfaces and libraries).

Imports:

- **Libraries**
  - LibKeep.sol
  - LibTokens.sol
  - LibDiamond.sol
  - LibLayoutSilo.sol
  - LibStorageBinder.sol

- **Interfaces**
  - IERC20.sol

State Variables

- **custom errors**

```
error AmountMismatch();
```

This error reverts when the amount parameter provided to depositEther() is not equal to the msg.value sent, indicating a failed transaction. see function definition below

- **Structs**

```
struct VaultInfo {
    address owner;
    uint256 weiBalance;
    uint256 lastPing;
    uint256 id;
    address backup;
    address[] inheritors;
}
```

This is the layout of the information stored for a vault in storage.

```
struct AllInheritorEtherAllocs {
    address inheritor;
    uint256 weiAlloc;
}
```

This is the layout of the information stored for each inheritor of a vault keeping track of the Ether balance in Wei allocated to an address.

- **Event**

```
event EthDeposited(uint256 _amount, uint256 _vaultID);
```

This event is emitted when a vault is successfully credited with ETH, indicating a successful transaction.

- **Read Functions**

```
function inspectVault() public view returns (VaultInfo memory info) {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    info.owner = vaultData.vaultOwner;
    info.weiBalance = address(this).balance;
    info.lastPing = vaultData.lastPing;
    info.id = vaultData.vaultID;
    info.backup = vaultData.backupAddress;
    info.inheritors = vaultData.inheritors;
}
```

This function returns the information of a vault from storage in the layout of the VaultInfo struct defined above.

```
function vaultOwner() public view returns (address) {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    return vaultData.vaultOwner;
}
```

This function returns the owner of a vault from storage.

```
function allEtherAllocations() public view returns (AllInheritorEtherAllocs[] memory eAllocs) {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    uint256 count = vaultData.inheritors.length;
    eAllocs = new AllInheritorEtherAllocs[](count);
    for (uint256 i; i < count; i++) {
        eAllocs[i].inheritor = vaultData.inheritors[i];
        eAllocs[i].weiAlloc = vaultData.inheritorWeishares[vaultData.inheritors[i]];
    }
}
```

This function returns an array of structs of all the inheritors of a vault where each element is in the layout of the AllInheritorEtherAllocs struct (an inheritors address and Wei balance).

```
function inheritorEtherAllocation(address _inheritor) public view returns (uint256 _allocatedEther) {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (!Guards._anInheritor(_inheritor)) {
        revert LibKeep.NotInheritor();
    }
    _allocatedEther = vaultData.inheritorWeishares[_inheritor];
}
```

This function reverts with an error (NotInheritor()) if Guards._anInheritor() returns a false when called with a supposed inheritors address. Returning false means the address is neither an address zero nor a true inheritor to the vault. Else the function returns the inheritors' shares in Wei stored in the inheritorWeishares mapping.

```
function getAllocatedEther() public view returns (uint256) {
    return LibKeep.getCurrentAllocatedEth();
}
```

This function calls another function getCurrentAllocatedEth() which returns a total of all the allocated eth of a vault.

```
function getUnallocatedEther() public view returns (uint256 unallocated_) {
    uint256 currentBalance = address(this).balance;
    if (currentBalance > 0) {
        unallocated_ = currentBalance - LibKeep.getCurrentAllocatedEth();
    }
}
```

This function returns the ether balance of a vault that is not allocated to any inheritor, by subtracting the result from calling getCurrentAllocatedEth() from the vault balance (address(this).balance).

```
function etherBalance() public view returns (uint256) {
    return address(this).balance;
}
```

This function returns the current ether balance of a vault.

- **Write functions**

```
function addInheritors(address[] calldata _newInheritors, uint256[] calldata _weiShare) external {
    Guards._onlyVaultOwnerOrOrigin();
    LibKeep._addInheritors(_newInheritors, _weiShare);
}
```

This function adds inheritor(s) to a vault, the function is guarded with _onlyVaultOwnerOrOrigin() which permits only the vault owner to add inheritors to owned vault, if the caller is the true vault owner _addInheritors() is called which does the heavy lifting, see function definition below.

```
function removeInheritors(address[] calldata _inheritors) external {
    Guards._onlyVaultOwner();
    LibKeep._removeInheritors(_inheritors);
}
```

Similar to the addInheritors() function above this function removes inheritor(s) from a vault, the function is also guarded with _onlyVaultOwner() which permits only the vault owner to remove inheritors from the owned vault, if the caller is the true vault owner _removeInheritors() is called which does the heavy lifting, see function definition below.

```
function depositEther(uint256 _amount) external payable {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (_amount != msg.value) {
        revert AmountMismatch();
    }
    emit EthDeposited(_amount, vaultData.vaultID);
}
```

This function handles ether deposit to a vault, the EthDeposited() event is emitted with the amount deposited, and the vault ID deposited to if the amount of ether to deposit is equal to the msg.value else it will revert with AmountMismatch() error.

```
function withdrawEther(uint256 _amount, address _to) external {
    Guards._onlyVaultOwner();
    LibKeep._withdrawEth(_amount, _to);
}
```

is guarded with the _onlyVaultOwner() function which ensures that only the owner of a vault can withdraw from it, if the check passes _withdrawEth() is called with the amount to withdraw and address to withdraw to. See the function definition below.

```
function allocateEther(address[] calldata _inheritors, uint256[] calldata _ethShares) external {
    Guards._onlyVaultOwner();
    LibKeep._allocateEther(_inheritors, _ethShares);
}
```

This function allocates ether to one or more inheritor(s) at a time, it is also guarded with _onlyVaultOwner() which permits only the vault owner to allocate ethers shares to inheritors of the owned vault, if the caller is the true vault owner _allocateEther() is called which does the heavy lifting, see function definition below.

```
function allocateERC20Tokens(address token, address[] calldata _inheritors, uint256[] calldata _shares) external {
    Guards._onlyVaultOwner();
    LibKeep._allocateERC20Tokens(token, _inheritors, _shares);
}
```

This function allocates erc20 tokens to one or more inheritor(s) at a time, it is also guarded with _onlyVaultOwner() which permits only the vault owner to allocate erc20 shares to inheritors of the owned vault, if the caller is the true vault owner _allocateERC20Tokens() is called which does the heavy lifting, see function definition below.

```
function allocateERC721Tokens(address token, address[] calldata _inheritors, uint256[] calldata _tokenIDs)
    external
{
    Guards._onlyVaultOwner();
    LibKeep._allocateERC721Tokens(token, _inheritors, _tokenIDs);
}
```

This function allocates a specific ERC721 token to specific inheritor(s) at a time, it is also guarded with _onlyVaultOwner() which permits only the vault owner to allocate ERC721 shares to inheritors of the owned vault, if the caller is the true vault owner _allocateERC721Tokens() is called which does the heavy lifting, see function definition below.

```
function allocateERC1155Tokens(
    address token,
    address[] calldata _inheritors,
    uint256[] calldata _tokenIDs,
    uint256[] calldata _amounts
)
    external
{
    Guards._onlyVaultOwner();
    LibKeep._allocateERC1155Tokens(token, _inheritors, _tokenIDs, _amounts);
}
```

This function allocates a specific ERC1155 token and a quantity of the token to specific inheritor(s) at a time, it is also guarded with _onlyVaultOwner() which permits only the vault owner to allocate ERC1155 shares to inheritors of the owned vault, if the caller is the true vault owner _allocateERC1155Tokens() is called which does the heavy lifting, see function definition below.

```
function transferOwnership(address _newVaultOwner) public {
    Guards._onlyVaultOwner();
    LibKeep._transferOwnerShip(_newVaultOwner);
}
```

This function transfers the owner of a vault to a new address, it is also guarded with _onlyVaultOwner() which permits only the vault owner to transfer ownership of the owned vault to a new owner if the caller is the true vault owner _transferOwnerShip() is called which does the heavy lifting, see function definition below.

```
function transferBackup(address _newBackupAddress) public {
    Guards._onlyVaultOwnerOrOriginOrBackup();
    LibKeep._transferBackup(_newBackupAddress);
}
```

This function changes the backup address of a vault to a new address, it is also guarded with _onlyVaultOwner() which permits only the vault owner to change the backup address of the owned vault to a new backup address if the caller is the true vault owner _transferBackup() is called which does the heavy lifting, see function definition below.

```
function claimOwnership(address _newBackupAddress) public {
    Guards._enforceIsBackupAddress();
    LibKeep._claimOwnership(_newBackupAddress);
}
```

This function allows a backup address to claim ownership of a vault if the current owner has not interacted with the vault for 6months or more, this function is guarded with _enforceIsBackupAddress(), which ensures that the caller (msg.sender) is the backup address for the given vault else it reverts the claim process.

The claim process continues by calling the _claimOwnership() with a new backup address, setting the owner of the vault to the caller and the vault backup address to the new backup address.

see _enforceIsBackupAddress() and this _claimOwnership() for more in-depth details.

```
function claimAllAllocations() external {
    LibKeep._claimAll();
}
```

This function lets an inheritor claim all the assets allotted to him in a vault, this function has several guards

_anInheritor() ensures that a caller is a true inheritor, _activeInheritor() returns true if an address is an active inheritor, _expired() reverts the claim process if the vault's last ping is less than or equal to 6 months, _notClaimed() ensures a particular inheritor has not claimed their allotted assets.

```
function ping() external {
    Guards._onlyVaultOwner();
    LibKeep._ping();
}
```

This function sets the last ping of a vault, this simply sets the last time a vault was interacted with by calling the _ping() function.

LibKeep.sol

- **Imported Libraries**
  - LibVaultStorage.sol
  - LibDiamond.sol
  - LibKeepHelpers.sol
  - LibLayoutSilo.sol
  - LibStorageBinder.sol

- **Imported Interfaces**
  - IERC20.sol
  - IERC721.sol
  - IERC1155.sol

- **Constants**

```
bytes4 constant ERC1155_ACCEPTED = 0xf23a6e61;
bytes4 constant ERC1155_BATCH_ACCEPTED = 0xbc197c81;
bytes4 constant ERC721WithCall = 0xb88d4fde;
```

- **Events**

```
event VaultPinged(uint256 lastPing, uint256 vaultID);
event InheritorsAdded(address[] newInheritors, uint256 vaultID);
event InheritorsRemoved(address[] inheritors, uint256 vaultID);
event EthAllocated(address[] inheritors, uint256[] amounts, uint256 vaultID);

event ERC20TokenWithdrawal(address token, uint256 amount, address to, uint256 vaultID);

event ERC721TokenWIthdrawal(address token, uint256 tokenID, address to, uint256 vaultID);
event ERC1155TokenWithdrawal(address token, uint256 tokenID, uint256 amount, address to, uint256 vaultID);
event ERC20ErrorHandled(address);
event ERC721ErrorHandled(uint256 _failedTokenId, string reason);

event ERC20TokensAllocated(address indexed token, address[] inheritors, uint256[] amounts, uint256 vaultID);
event ERC721TokensAllocated(address indexed token, address inheritor, uint256 tokenID, uint256 vaultID);
event ERC1155TokensAllocated(
    address indexed token, address inheritor, uint256 tokenID, uint256 amount, uint256 vaultID
);
event OwnershipTransferred(address indexed previousOwner, address indexed newOwner, uint256 vaultID);
event BackupTransferred(address indexed previousBackup, address indexed newBackup, uint256 vaultID);
event EthClaimed(address indexed inheritor, uint256 _amount, uint256 vaultID);

event ERC20TokensClaimed(address indexed inheritor, address indexed token, uint256 amount, uint256 vaultID);

event ERC721TokenCla
event ERC1155TokensC
    address indexed
);
```

```
//owner check is in external fn
function _ping() internal {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    vaultData.lastPing = block.timestamp;
    emit VaultPinged(block.timestamp, _vaultID());
}
```

```
ID, uint256 vaultID);

int256 vaultID
```

- **Errors**

```
error LengthMismatch();
error ActiveInheritor();
error NotEnoughEtherToAllocate(uint256);
error EmptyArray();
error NotInheritor();
error EtherAllocationOverflow(uint256 overflow);
error TokenAllocationOverflow(address token, uint256 overflow);
error InactiveInheritor();
error InsufficientEth();
error InsufficientTokens();
error NoAllocatedTokens();
error NotERC721Owner();
```

- **Functions**

```
//owner check is in external fn
function _ping() internal {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    vaultData.lastPing = block.timestamp;
    emit VaultPinged(block.timestamp, _vaultID());
}
```

This function updates the lastPing vaultData in storage with the current block. timestamp whenever the vault storage is modified and it emits an event VaultPinged() with the current block.timestamp and vaultId.

```
function getCurrentAllocatedEth() internal view returns (uint256) {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    uint256 totalEthAllocated;
    for (uint256 x; x < vaultData.inheritors.length; x++) {
        totalEthAllocated += vaultData.inheritorWeishares[vaultData.inheritors[x]];
    }
    return totalEthAllocated;
}
```

This function returns a total of all the ETH shares allocated to inheritors in a vault.

```
function getCurrentAllocatedTokens(address _token) internal view returns (uint256) {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    uint256 totalTokensAllocated;
    for (uint256 x; x < vaultData.inheritors.length; x++) {
        totalTokensAllocated += vaultData.inheritorTokenShares[vaultData.inheritors[x]][_token];
    }
    return totalTokensAllocated;
}
```

This function returns a total of all the token shares for a particular token allocated to inheritors in a vault.

- **Functions**

```
function getCurrentAllocated1155tokens(address _token, uint256 _tokenID) internal view returns (uint256 alloc_) {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    for (uint256 x; x < vaultData.inheritors.length; x++) {
        alloc_ += vaultData.inheritorERC1155TokenAllocations[vaultData.inheritors[x]][_token][_tokenID];
    }
}
```

This function returns a total of all the token shares for a particular ERC1155 allocated to inheritors in a vault.

```
function _vaultID() internal view returns (uint256 vaultID_) {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    vaultID_ = vaultData.vaultID;
}
```

This function returns the vaultId of a particular vault.

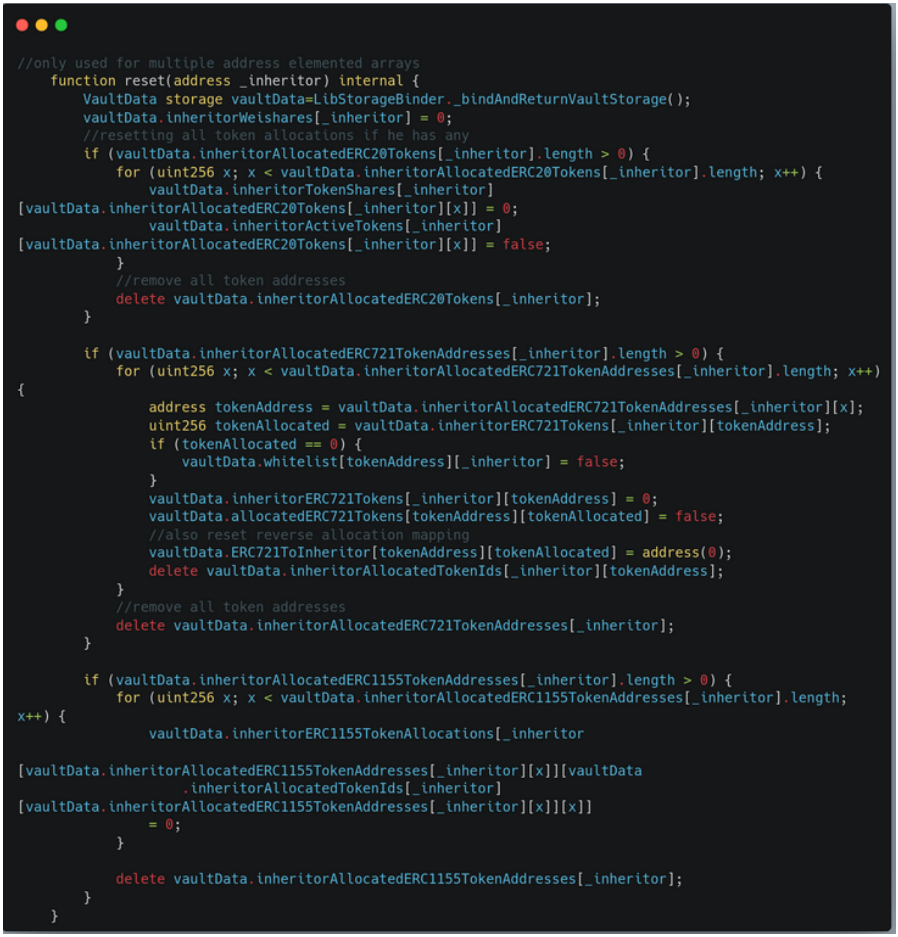
```
function _resetClaimed(address _inheritor) internal {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    vaultData.inheritorWeishares[_inheritor] = 0;
    //resetting all token allocations if he has any
    if (vaultData.inheritorAllocatedERC20Tokens[_inheritor].length > 0) {
        //remove all token addresses
        delete vaultData.inheritorAllocatedERC20Tokens[_inheritor];
    }

    if (vaultData.inheritorAllocatedERC721TokenAddresses[_inheritor].length > 0) {
        delete vaultData.inheritorAllocatedERC721TokenAddresses[_inheritor];
    }

    if (vaultData.inheritorAllocatedERC1155TokenAddresses[_inheritor].length > 0) {
        delete vaultData.inheritorAllocatedERC1155TokenAddresses[_inheritor];
    }
}
```

This function reset the balances of ETH, ERC721, ERC1155, and ERC20 tokens of an inheritor to zero after they have claimed.

- **Functions**



```
//only used for multiple address elemented arrays
    function reset(address _inheritor) internal {
        VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
        vaultData.inheritorWeishares[_inheritor] = 0;
        //resetting all token allocations if he has any
        if (vaultData.inheritorAllocatedERC20Tokens[_inheritor].length > 0) {
            for (uint256 x; x < vaultData.inheritorAllocatedERC20Tokens[_inheritor].length; x++) {
                vaultData.inheritorTokenShares[_inheritor]
[vaultData.inheritorAllocatedERC20Tokens[_inheritor][x]] = 0;
                vaultData.inheritorActiveTokens[_inheritor]
[vaultData.inheritorAllocatedERC20Tokens[_inheritor][x]] = false;
            }
            //remove all token addresses
            delete vaultData.inheritorAllocatedERC20Tokens[_inheritor];
        }

        if (vaultData.inheritorAllocatedERC721TokenAddresses[_inheritor].length > 0) {
            for (uint256 x; x < vaultData.inheritorAllocatedERC721TokenAddresses[_inheritor].length; x++)
{
                address tokenAddress = vaultData.inheritorAllocatedERC721TokenAddresses[_inheritor][x];
                uint256 tokenAllocated = vaultData.inheritorERC721Tokens[_inheritor][tokenAddress];
                if (tokenAllocated == 0) {
                    vaultData.whitelist[tokenAddress][_inheritor] = false;
                }
                vaultData.inheritorERC721Tokens[_inheritor][tokenAddress] = 0;
                vaultData.allocatedERC721Tokens[tokenAddress][tokenAllocated] = false;
                //also reset reverse allocation mapping
                vaultData.ERC721ToInheritor[tokenAddress][tokenAllocated] = address(0);
                delete vaultData.inheritorAllocatedTokenIds[_inheritor][tokenAddress];
            }
            //remove all token addresses
            delete vaultData.inheritorAllocatedERC721TokenAddresses[_inheritor];
        }

        if (vaultData.inheritorAllocatedERC1155TokenAddresses[_inheritor].length > 0) {
            for (uint256 x; x < vaultData.inheritorAllocatedERC1155TokenAddresses[_inheritor].length;
x++) {
                vaultData.inheritorERC1155TokenAllocations[_inheritor

[vaultData.inheritorAllocatedERC1155TokenAddresses[_inheritor][x]][vaultData
                    .inheritorAllocatedTokenIds[_inheritor]
[vaultData.inheritorAllocatedERC1155TokenAddresses[_inheritor][x]][x]]
                = 0;
            }

            delete vaultData.inheritorAllocatedERC1155TokenAddresses[_inheritor];
        }
    }
```

This function also reset the balances of ETH, ERC721, ERC1155, and ERC20 tokens of an inheritor but this is only used for multiple address elemented arrays.

- **Functions**

```
function _addInheritors(address[] calldata _newInheritors, uint256[] calldata _weiShare) internal {
    if (_newInheritors.length == 0 || _weiShare.length == 0) {
        revert EmptyArray();
    }
    if (_newInheritors.length != _weiShare.length) {
        revert LengthMismatch();
    }
    Guards._notExpired();
    uint256 total;
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    for (uint256 k; k < _newInheritors.length; k++) {
        total += _weiShare[k];

        if (vaultData.activeInheritors[_newInheritors[k]]) {
            revert ActiveInheritor();
        }
        //append the inheritors for a vault
        vaultData.inheritors.push(_newInheritors[k]);
        vaultData.activeInheritors[_newInheritors[k]] = true;
        //    if (total + allocated > address(this).balance)
        //        revert NotEnoughEtherToAllocate(address(this).balance);
        //    vaultData.inheritorWeishares[_newInheritors[k]] = _weiShare[k];
    }
    _allocateEther(_newInheritors, _weiShare);

    _ping();
    emit InheritorsAdded(_newInheritors, _vaultID());
    emit EthAllocated(_newInheritors, _weiShare, _vaultID());
}
```

This Function Adds inheritors to a vault, with their corresponding Wei balance. it ensures the array if inheritors to be added is not empty, it is guarded with _notExpired() which reverts the whole adding process if the last ping time is greater than 24 weeks (six months). it calls the _ping() function after this operation to update the lastPing in storage. It also emits two events InheritorsAdded() and EthAllocated()

- **Functions**

```
function _removeInheritors(address[] calldata _inheritors) internal {
    if (_inheritors.length == 0) {
        revert EmptyArray();
    }
    Guards._notExpired();

    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    for (uint256 k; k < _inheritors.length; k++) {
        if (!vaultData.activeInheritors[_inheritors[k]]) {
            revert NotInheritor();
        }
        vaultData.activeInheritors[_inheritors[k]] = false;
        //pop out the address from the array
        LibKeepHelpers.removeAddress(vaultData.inheritors, _inheritors[k]);
        reset(_inheritors[k]);
    }
    _ping();
    emit InheritorsRemoved(_inheritors, _vaultID());
}
```

This Function Removes inheritors from a vault, with their corresponding Wei balance. it ensures the array of inheritors to be removed is not empty, it is guarded with _notExpired() which reverts the whole removing process if the last ping time is greater than 24 weeks (six months). it calls the _ping() function after this operation to update the lastPing in storage. It also emits an event InheritorRemoved()

- **Functions**

```
function _allocateERC20Tokens(address token, address[] calldata _inheritors, uint256[] calldata
_shares) internal {
    if (_inheritors.length == 0 || _shares.length == 0) {
        revert EmptyArray();
    }
    if (_inheritors.length != _shares.length) {
        revert LengthMismatch();
    }
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    for (uint256 k; k < _inheritors.length; k++) {
        if (!Guards._anInheritor(_inheritors[k])) {
            revert NotInheritor();
        }
        if (!Guards._activeInheritor(_inheritors[k])) {
            revert InactiveInheritor();
        }
        vaultData.inheritorTokenShares[_inheritors[k]][token] = _shares[k];
        if (!vaultData.inheritorActiveTokens[_inheritors[k]][token] && _shares[k] > 0) {
            vaultData.inheritorAllocatedERC20Tokens[_inheritors[k]].push(token);
            vaultData.inheritorActiveTokens[_inheritors[k]][token] = true;
        }
        //if allocation is being reduced to zero
        if (_shares[k] == 0) {
            LibKeepHelpers.removeAddress(vaultData.inheritorAllocatedERC20Tokens[_inheritors[k]],
token);
            //double-checking
            vaultData.inheritorActiveTokens[_inheritors[k]][token] = false;
        }
        //finally check that limit isn't exceeded
        //get vault token balance
        uint256 currentBalance = IERC20(token).balanceOf(address(this));
        if (getCurrentAllocatedTokens(token) > currentBalance) {
            revert TokenAllocationOverflow(token, getCurrentAllocatedTokens(token) - currentBalance);
        }
    }
    _ping();
    emit ERC20TokensAllocated(token, _inheritors, _shares, _vaultID());
}
```

This Function sets token allocations for a particular ERC20 for a list of inheritors. it ensures the array of inheritors and array of shears provided have the same length and they are not empty arrays, it is guarded with _aninheritor() and _activeInheritor() which returns true for an active inheritor and false otherwise, if allocation is reduced to zero, address is removed. it calls the _ping() function after this operation to update the lastPing in storage. It also emits an event EthAllocated().

- **Functions**

```
    function _allocateERC721Tokens(address _token, address[] calldata _inheritors, uint256[] calldata
_tokenIDs)
        internal
  {
      if (_inheritors.length == 0 || _tokenIDs.length == 0) {
          revert EmptyArray();
      }
      if (_inheritors.length != _tokenIDs.length) {
          revert LengthMismatch();
      }
      VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
      for (uint256 k; k < _inheritors.length; k++) {
          if (!Guards._anInheritorOrZero(_inheritors[k])) {
              revert NotInheritor();
          }
          if (!Guards._activeInheritor(_inheritors[k])) {
              revert InactiveInheritor();
          }
          //short-circuit
          if (vaultData.ERC721ToInheritor[_token][_tokenIDs[k]] == _inheritors[k]) {
              continue;
          }
          //confirm ownership
          try IERC721(_token).ownerOf(_tokenIDs[k]) returns (address owner) {
              if (owner == address(this)) {
                  if (vaultData.allocatedERC721Tokens[_token][_tokenIDs[k]]) {
                      address current = vaultData.ERC721ToInheritor[_token][_tokenIDs[k]];
                      //if it is being allocated to someone else
                      if (current != _inheritors[k] && current != address(0) && _inheritors[k] !=
                        address(0)) {
                          //Might add an Unallocation event
                          vaultData.whitelist[_token][current] = false;

LibKeepHelpers.removeUint(vaultData.inheritorAllocatedTokenIds[current[_token], _tokenIDs[k]);
                          //if no tokens remain for that address
                          if (vaultData.inheritorAllocatedTokenIds[current][_token].length == 0) {
                              //remove the address

 LibKeepHelpers.removeAddress(vaultData.inheritorAllocatedERC721TokenAddresses[current], _token);
                          }
                      }
                      //if it is being unallocated
                      if (_inheritors[k] == address(0)) {
                          vaultData.allocatedERC721Tokens[_token][_tokenIDs[k]] = false;
                          LibKeepHelpers.removeUint(vaultData.inheritorAllocatedTokenIds[current]
[_token], _tokenIDs[k]);

                          if (vaultData.inheritorAllocatedTokenIds[_inheritors[k]][_token].length == 0)
{

 LibKeepHelpers.removeAddress(vaultData.inheritorAllocatedERC721TokenAddresses[current], _token);
                          }
                      }
                  } else {
                      vaultData.allocatedERC721Tokens[_token][_tokenIDs[k]] = true;
                  }
                  vaultData.ERC721ToInheritor[_token][_tokenIDs[k]] = _inheritors[k];
                  if (vaultData.inheritorAllocatedTokenIds[_inheritors[k]][_token].length == 0) {
                      vaultData.inheritorAllocatedERC721TokenAddresses[_inheritors[k]].push(_token);
                  }

                  vaultData.inheritorAllocatedTokenIds[_inheritors[k]][_token].push(_tokenIDs[k]);

                  if (_tokenIDs[k] == 0) {
                      vaultData.whitelist[_token][_inheritors[k]] = true;
                  }
                  //   vaultData.inheritorERC721Tokens[_inheritors[k]][_token] = _tokenIDs[k];
                  emit ERC721TokensAllocated(_token, _inheritors[k], _tokenIDs[k], _vaultID());
              }
              if (owner != address(this)) {
                  emit ERC721ErrorHandled(_tokenIDs[k], "Not_Owner");
                  continue;
              }
          } catch Error(string memory r) {
              emit ERC721ErrorHandled(_tokenIDs[k], r);
              continue;
          }
      }
      _ping();
  }
```

- **Functions**

This Function sets token allocations for a particular ERC21 for a list of inheritors. it ensures the array of inheritors and array of shears provided have the same length and they are not empty arrays, it is guarded with _aninheritorOrZero() which returns true for a true inheritor or a zero address and _activeInheritor() which returns true for an active inheritor and false otherwise, if the caller of this function is not the contract address the process is reversed with ERC721ErrorHandled() error. It calls the _ping() function after this operation to update the lastPing in storage. It also emits an event EthAllocated().

```
    function _allocateERC1155Tokens(
        address _token,
        address[] calldata _inheritors,
        uint256[] calldata _tokenIDs,
        uint256[] calldata _amounts
    )
        internal
    {
        if (_inheritors.length == 0 || _tokenIDs.length == 0) {
            revert EmptyArray();
        }
        if (_inheritors.length != _tokenIDs.length) {
            revert LengthMismatch();
        }
        if (_inheritors.length != _amounts.length) {
            revert LengthMismatch();
        }
        VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
        for (uint256 i; i < _inheritors.length; i++) {
            if (!Guards._anInheritor(_inheritors[i])) {
                revert NotInheritor();
            }
            if (!Guards._activeInheritor(_inheritors[i])) {
                revert InactiveInheritor();
            }
            vaultData.inheritorERC1155TokenAllocations[_inheritors[i]][_token][_tokenIDs[i]] =
_amounts[i];
            //if id is just being added
            if (!LibKeepHelpers._inUintArray(vaultData.inheritorAllocatedTokenIds[_inheritors[i]]
[_token], _tokenIDs[i])) {
                vaultData.inheritorAllocatedTokenIds[_inheritors[i]][_token].push(_tokenIDs[i]);
            }
            //if address is just being added
            if
(!LibKeepHelpers._inAddressArray(vaultData.inheritorAllocatedERC1155TokenAddresses[_inheritors[i]],
_token)) {
                vaultData.inheritorAllocatedERC1155TokenAddresses[_inheritors[i]].push(_token);
            }
            //if tokens are being unallocated
            if (_amounts[i] == 0) {
                LibKeepHelpers.removeUint(vaultData.inheritorAllocatedTokenIds[_inheritors[i]][_token],
_tokenIDs[i]);
            }
            //if no tokens for the token address remain
            if (vaultData.inheritorAllocatedTokenIds[_inheritors[i]][_token].length == 0) {

 LibKeepHelpers.removeAddress(vaultData.inheritorAllocatedERC1155TokenAddresses[_inheritors[i]],
_token);
            }
            //confirm numbers
            uint256 allocated = getCurrentAllocated1155tokens(_token, _tokenIDs[i]);
            uint256 available = IERC1155(_token).balanceOf(address(this), _tokenIDs[i]);
            if (allocated > available) {
                revert TokenAllocationOverflow(_token, allocated - available);
            }

            emit ERC1155TokensAllocated(_token, _inheritors[i], _tokenIDs[i], _amounts[i], _vaultID());
        }

        _ping();
    }
```

- **Functions**

This Function sets token allocations for a particular ERC1155 for a list of inheritors. it ensures the array of inheritors, array of tokenIds and array of amount provided have the same length and they are not empty arrays, it is guarded with _aninheritor() which returns true for a true inheritor and _activeInheritor() which returns true for an active inheritor and false otherwise, if no tokens for the token address remain the token address is removed from the list allocated ERC1155 address. It calls the _ping() function after this operation to update the lastPing in storage. It also emits an event ERC1155TokensAllocated().

- **_withdrawEth function**

```
function _withdrawEth(uint256 _amount, address _to) internal {
    //confirm free eth is sufficient
    uint256 allocated = getCurrentAllocatedEth();
    if (address(this).balance >= allocated) {
        if (address(this).balance - allocated < _amount) {
            revert InsufficientEth();
        }
        (bool success,) = _to.call{value: _amount}("");
        assert(success);
    } else {
        revert InsufficientEth();
    }
}
```

This function is used to withdraw unallocated ETH balance.

- **_withdrawERC20Tokens Functions**

```
function _withdrawERC20Tokens(address[] calldata _tokenAdds, uint256[] calldata _amounts, address _to)
internal {
        if (_tokenAdds.length == 0 || _amounts.length == 0) {
            revert EmptyArray();
        }
        if (_tokenAdds.length != _amounts.length) {
            revert LengthMismatch();
        }
        // VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
        for (uint256 x; x < _tokenAdds.length; x++) {
            address token = _tokenAdds[x];
            uint256 amount = _amounts[x];
            uint256 availableTokens = getCurrentAllocatedTokens(token);
            uint256 currentBalance = IERC20(token).balanceOf(address(this));
            bool success;
            if (currentBalance >= availableTokens) {
                if (currentBalance - availableTokens < _amounts[x]) {
                    revert InsufficientTokens();
                }
                //for other errors caused by malformed tokens
                try IERC20(token).transfer(_to, amount) {
                    success;
                } catch {
                    if (success) {
                        emit ERC20TokenWithdrawal(token, amount, _to, _vaultID());
                    } else {
                        emit ERC20ErrorHandled(token);
                    }
                }
            } else {
                revert InsufficientTokens();
            }
        }
        _ping();
    }
```

This function is used to withdraw unallocated ERC20 Tokens balance.

- **_withdrawERC20Token Functions**

```
function _withdrawERC20Token(address _token, uint256 _amount, address _to) internal {
        uint256 availableTokens = getCurrentAllocatedTokens(_token);
        uint256 currentBalance = IERC20(_token).balanceOf(address(this));
        bool success;
        if (currentBalance >= availableTokens) {
            if (currentBalance - availableTokens < _amount) {
                revert InsufficientTokens();
            }
            try IERC20(_token).transfer(_to, _amount) {
                success;
            } catch {
                if (success) {
                    emit ERC20TokenWithdrawal(_token, _amount, _to, _vaultID());
                } else {
                    emit ERC20ErrorHandled(_token);
                }
            }
        } else {
            revert InsufficientTokens();
        }

        _ping();
    }
```

This function is used to withdraw unallocated ERC20 Token balance.

- **_withdrawERC721Token Functions**

```
    function _withdrawERC721Token(address _token, uint256 _tokenID, address _to) internal {
        if (IERC721(_token).ownerOf(_tokenID) != address(this)) {
            revert NotERC721Owner();
        }
        VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
        if (vaultData.allocatedERC721Tokens[_token][_tokenID]) {
            revert("UnAllocate Token First");
        }
        try IERC721(_token).safeTransferFrom(address(this), _to, _tokenID) {}
        catch {
            string memory reason;
            if (bytes(reason).length == 0) {
                emit ERC721TokenWIthdrawal(_token, _tokenID, _to, _vaultID());
            } else {
                emit ERC20ErrorHandled(_token);
            }
        }
    }
```

This function is used to withdraw the ERC721 Token balance.

- **_withdrawERC1155Token Functions**

```
    function _withdrawERC1155Token(address _token, uint256 _tokenID, uint256 _amount, address _to)
internal {
        uint256 allocated = getCurrentAllocated1155tokens(_token, _tokenID);
        uint256 balance = IERC1155(_token).balanceOf(address(this), _tokenID);
        if (balance < _amount) {
            revert InsufficientTokens();
        }

        if (balance - allocated < _amount) {
            revert("UnAllocate TokensFirst");
        }
        IERC1155(_token).safeTransferFrom(address(this), _to, _tokenID, _amount, "");
        emit ERC1155TokenWithdrawal(_token, _tokenID, _amount, _to, _vaultID());
    }
```

This function is used to withdraw unallocated ERC1155 token.

- **_transferOwnerShip Functions**

```
function _transferOwnerShip(address _newOwner) internal {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    address prevOwner = vaultData.vaultOwner;
    vaultData.vaultOwner = _newOwner;
    emit OwnershipTransferred(prevOwner, _newOwner, _vaultID());
}
```

This function is used to transfer ownership of a vault.

- **_transferBackup Functions**

```
function _transferBackup(address _newBackupAddress) internal {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    address prevBackup = vaultData.backupAddress;
    vaultData.backupAddress = _newBackupAddress;
    emit BackupTransferred(prevBackup, _newBackupAddress, _vaultID());
}
```

This function is used to change the backup address.

- **_claimOwnership Functions**

```
function _claimOwnership(address _newBackup) internal {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    Guards._expired();
    address prevOwner = vaultData.vaultOwner;
    address prevBackup = vaultData.backupAddress;
    assert(prevOwner != _newBackup);
    vaultData.vaultOwner = msg.sender;
    vaultData.backupAddress = _newBackup;
    emit OwnershipTransferred(prevOwner, msg.sender, _vaultID());
    emit BackupTransferred(prevBackup, _newBackup, _vaultID());
}
```

This function allows a backup address to claim ownership of a vault with the condition that the vault has reached expiry ie the last ping of the vault is up to or more than 6months

- **_claimERC20Tokens Functions**

```
function _claimERC20Tokens() internal {
    // Guards._anInheritor(msg.sender);
    // Guards._activeInheritor(msg.sender);
    // Guards._expired();
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    uint256 tokens = vaultData.inheritorAllocatedERC20Tokens[msg.sender].length;
    if (tokens > 0) {
        for (uint256 i; i < tokens; i++) {
            address token = vaultData.inheritorAllocatedERC20Tokens[msg.sender][i];
            if (token == address(0)) {
                continue;
            }
            uint256 amountToClaim = vaultData.inheritorTokenShares[msg.sender][token];
            if (amountToClaim > 0) {
                //reset storage
                vaultData.inheritorTokenShares[msg.sender][token] = 0;
                IERC20(token).transfer(msg.sender, amountToClaim);
                emit ERC20TokensClaimed(msg.sender, token, amountToClaim, _vaultID());
            }
        }
    }
}
```

This function helps inheritors to claim the erc20 tokens allotted to them.

- **_claimERC721Tokens Functions**

```
function _claimERC721Tokens() internal {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    uint256 tokens = vaultData.inheritorAllocatedERC721TokenAddresses[msg.sender].length;
    if (tokens > 0) {
        for (uint256 i; i < tokens; i++) {
            address token = vaultData.inheritorAllocatedERC721TokenAddresses[msg.sender][i];
            if (token == address(0)) {
                continue;
            }
            uint256 tokensToClaim = vaultData.inheritorAllocatedTokenIds[msg.sender][token].length;
            if (tokensToClaim > 0) {
                for (uint256 j; j < tokensToClaim; j++) {
                    uint256 tokenID = vaultData.inheritorAllocatedTokenIds[msg.sender][token][j];
                    if (tokenID == 0) {
                        //check for whitelist
                        if (vaultData.whitelist[token][msg.sender]) {
                            vaultData.whitelist[token][msg.sender] = false;
                            IERC721(token).transferFrom(address(this), msg.sender, 0);
                            emit ERC721TokenClaimed(msg.sender, token, 0, _vaultID());
                        }
                    } else {
                        //test thorougly for array overflows
                        vaultData.inheritorAllocatedTokenIds[msg.sender][token][j] = 0;
                        IERC721(token).transferFrom(address(this), msg.sender, tokenID);
                        emit ERC721TokenClaimed(msg.sender, token, tokenID, _vaultID());
                    }
                }
            }
        }
    }
}
```

This function helps inheritors to claim the ERC721 tokens allotted to them.

- **_claimERC1155Tokens Functions**

```
function _claimERC1155Tokens() internal {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    uint256 tokens = vaultData.inheritorAllocatedERC1155TokenAddresses[msg.sender].length;
    if (tokens > 0) {
        for (uint256 i; i < tokens; i++) {
            address token = vaultData.inheritorAllocatedERC1155TokenAddresses[msg.sender][i];
            if (token == address(0)) {
                continue;
            }
            uint256 noOfTokenIds = vaultData.inheritorAllocatedTokenIds[msg.sender][token].length;
            if (noOfTokenIds > 0) {
                for (uint256 k; k < noOfTokenIds; k++) {
                    uint256 tokenID = vaultData.inheritorAllocatedTokenIds[msg.sender][token][k];
                    uint256 amount = vaultData.inheritorERC1155TokenAllocations[msg.sender][token]
[tokenID];

                    if (amount > 0) {
                        vaultData.inheritorERC1155TokenAllocations[msg.sender][token][tokenID] = 0;
                        IERC1155(token).safeTransferFrom(address(this), msg.sender, tokenID,
amount, "");

                        emit ERC1155TokensClaimed(msg.sender, token, 1, amount, _vaultID());
                    }
                }
            }
        }
    }
}
```

This function helps inheritors to claim the ERC1155 tokens allotted to them.

- **_claimAll Functions**

```
function _claimAll() internal {
    Guards._anInheritor(msg.sender);
    Guards._activeInheritor(msg.sender);
    Guards._expired();
    Guards._notClaimed(msg.sender);
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (vaultData.inheritorWeishares[msg.sender] > 0) {
        uint256 amountToClaim = vaultData.inheritorWeishares[msg.sender];
        //reset storage
        vaultData.inheritorWeishares[msg.sender] == 0;
        (bool success,) = msg.sender.call{value: amountToClaim}("");
        assert(success);

        emit EthClaimed(msg.sender, amountToClaim, _vaultID());
    }
    //claim ERC20 tokens..if any
    _claimERC20Tokens();
    //claim ERC721 Tokens if any
    _claimERC721Tokens();
    //claim ERC1155 Tokens if any
    _claimERC1155Tokens();

    //cleanup
    LibKeepHelpers.removeAddress(vaultData.inheritors, msg.sender);
    //clear storage
    //test thorougly
    _resetClaimed(msg.sender);
}
```

This function helps inheritors to claim all asset allotted to them

Guards Library Functions

- **_onlyVaultOwner()**

This function acts as an onlyOwner modifier that checks if the caller of a function is the true owner of the vault else operation reverts

- **_onlyVaultOwnerOrOrigin Function**

This function acts as an _onlyVaultOwnerOrOrigin modifier that checks if the caller of a function is either the true owner of the vault or the tx.origin else operation reverts.

- **_onlyVaultOwnerOrOriginOrBackup Functiion**

```
function _onlyVaultOwnerOrOriginOrBackup() internal view {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (
        tx.origin != vaultData.vaultOwner && msg.sender != vaultData.vaultOwner && msg.sender !=
           vaultData.backupAddress
            && tx.origin != vaultData.backupAddress
    ) {
        revert NoPermissions();
    }
}
```

This function acts as an _onlyVaultOwnerOrOriginOrBackup modifier that checks if the caller of a function is the true owner of the vault or the tx.origin or the backup address else operation reverts.

- **_onlyVaultOwnerOrBackup Function**

```
function _onlyVaultOwnerOrBackup() internal view {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (msg.sender != vaultData.backupAddress && msg.sender != vaultData.vaultOwner) {
        revert NotOwnerOrBackupAddress();
    }
}
```

This function acts as an _onlyVaultOwnerOrOrigin modifier that checks if the caller of a function is either the true owner of the vault or the Backup address else operation reverts.

- **_enforceIsBackupAddress Functiion**

```
function _enforceIsBackupAddress() internal view {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (msg.sender != vaultData.backupAddress) {
        revert NotBackupAddress();
    }
}
```

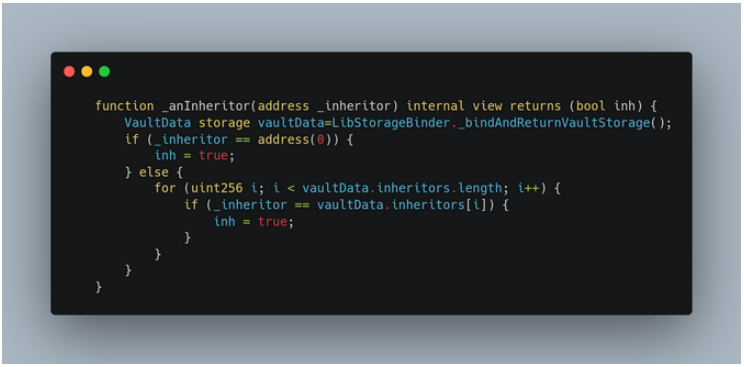
This function acts as an _enforceIsBackupAddress modifier that checks if the caller of a function is the backup address of the vault else the operation reverts.

- **_activeInheritor Function**

```
function _activeInheritor(address _inheritor) internal view returns (bool active_) {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (_inheritor == address(0)) {
        active_ = true;
    } else {
        active_ = (vaultData.activeInheritors[_inheritor]);
    }
}
```

This function checks if an inheritor is active in a vault if this function returns true the inheritor is allowed to proceed with the operation else operation reverts

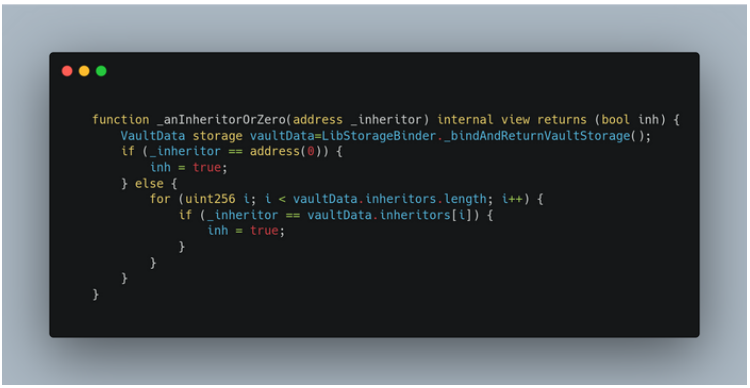
- **_anInheritor Functiion**



```
function _anInheritor(address _inheritor) internal view returns (bool inh) {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (_inheritor == address(0)) {
        inh = true;
    } else {
        for (uint256 i; i < vaultData.inheritors.length; i++) {
            if (_inheritor == vaultData.inheritors[i]) {
                inh = true;
            }
        }
    }
}
```

This function checks if a caller is a valid inheritor for a vault, it returns true if is valid and false otherwise

- **_anInheritorOrZero Function**



```
function _anInheritorOrZero(address _inheritor) internal view returns (bool inh) {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (_inheritor == address(0)) {
        inh = true;
    } else {
        for (uint256 i; i < vaultData.inheritors.length; i++) {
            if (_inheritor == vaultData.inheritors[i]) {
                inh = true;
            }
        }
    }
}
```

This function checks if a caller is a valid inheritor for a vault or a zero address, it returns true if is valid or zero address and false otherwise

- **_expired Functiion**

```
function _expired() internal view {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (block.timestamp - vaultData.lastPing <= 24 weeks) {
        revert NotExpired();
    }
}
```

This function checks if the last ping of a vault is less than or equal to 24 weeks, it reverts if this is true showing the vault is not epired yet but if the last ping is greater than 24 weeks the vault is expired and operation proceeds

- **_notExpired Function**

```
function _notExpired() internal view {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (block.timestamp - vaultData.lastPing > 24 weeks) {
        revert HasExpired();
    }
}
```

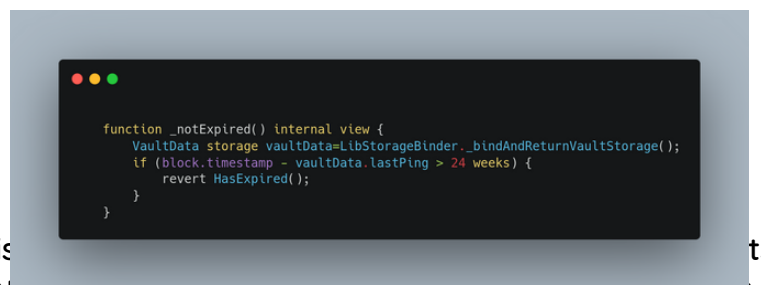
This function checks a vault has expired, and reverts if the last ping time of a vault has exceeded 24 weeks

- **_notClaimed Functiion**

```
function _notClaimed(address _inheritor) internal view {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (vaultData.claimed[_inheritor]) {
        revert Claimed();
    }
}
```

This function checks if an inheritor has claimed assets allotted to him, it reverts if the inheritor have claimed and proceeds if he hasent

- **_notExpired Function**



```
function _notExpired() internal view {
    VaultData storage vaultData=LibStorageBinder._bindAndReturnVaultStorage();
    if (block.timestamp - vaultData.lastPing > 24 weeks) {
        revert HasExpired();
    }
}
```

This ts if the last ping time of a vault has exceeded 24 weeks