CD SECURITY

For the Few Who Demand Perfection

AUDIT REPORT

Hyperlend

January 2026

Prepared by
radev_eth
zeroXchad

www.cdsecurity.io

Introduction

A time-boxed security review of the **Hyperlend** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Hyperlend**

`HplToken.sol`:

Upgradeable ERC20 + EIP-2612 permit token. On initialization it writes a "HyperCore deployer" address into a custom storage slot for external verification and mints the entire fixed supply (1B tokens) to a deterministic system address derived from hyperCoreIndex. After initialization, it behaves like a standard ERC20 with permit.

`MerkleDistributor`:

Merkle-based airdrop distributor with a claim deadline. Users prove their allocation via Merkle proof and receive tokens before endTime. Key feature: owner can set alternate recipients for specific accounts. Both the account and its registered recipient are authorized to trigger claims. Uses bit-packed mapping for gas-efficient claim tracking. After deadline, owner can withdraw unclaimed tokens

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

review commit hash - **846dc3dc6346c362e88da66d523e7e9471e988d1**

review commit hash - **a1756998353c82c408e0df0f433be80a60e1f283**

Scope

The following smart contracts were in scope of the audit:

- `HplToken.sol`
- `MerkleDistrbutonForContracts.sol`
- `MerkleDistributor.sol`
- `MerkleDistributorWithDeadline.sol`
- `IMerkleDistributor.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 0 issues
- Low & Info: 11 issues

Findings Summary

ID	Title	Severity	Status
[L-01]	Recipient mapping cannot be cleared once set in `MerkleDistributorForContracts.sol` smart contract	Low	Fixed
[I-01]	The incorrect RPC URL equality check in the deployment script for `HplToken.sol` can misdetect the network	Informational	Fixed
[I-02]	No getter function for stored `hyperCoreSpotDeployer` in `HplToken.sol` contract	Informational	Fixed
[I-03]	Missing zero-address/zero-bytes32 validation in `MerkleDistributorForContracts.sol` constructor	Informational	Fixed
[I-04]	`MerkleDistributorForContracts.sol#claim()` function wrongly emits recipient in event when no redirect is configured	Informational	Fixed
[I-05]	Lack of two step ownership transfer implementation	Informational	Acknowledged
[I-06]	OpenZeppelin library version 4.7.0 has known vulnerabilities	Informational	Acknowledged
[I-07]	HplToken makes use of Transparent Proxy	Informational	Acknowledged
[I-08]	Arbitrary recipient can be set by claimer	Informational	Fixed
[I-09]	`block.timestamp` equal to `endTime` instance is not handled	Informational	Acknowledged

ID	Title	Severity	Status
[I-10]	HplToken is not compliant with EIP-7201	Informational	Acknowledged

Detailed Findings

[L-01] Recipient mapping cannot be cleared once set in `MerkleDistributorForContracts.sol` smart contract

Severity

Impact: Low

Likelihood: Low

Description

`MerkleDistributorForContracts.sol#setRecipients()` function blocks `address(0)`:

```
    function setRecipients(address[] calldata _accounts, address[]
calldata _recipients) external onlyOwner {
        if (_accounts.length != _recipients.length) revert
InvalidLength();

        for (uint256 i = 0; i < _accounts.length; i++){
            if (_recipients[i] == address(0)) revert InvalidRecipient();
            recipients[_accounts[i]] = _recipients[i];
            emit RecipientSet(_accounts[i], _recipients[i]);
        }
    }
```

Once set, a recipient cannot be removed, only changed to another non-zero address, making it impossible to remove a previously configured recipient. Once a recipient is set for an account, all claims permanently redirect to that address and can never revert to direct claiming by the account.

If a wrong recipient is configured or the recipient becomes inaccessible, claims remain permanently redirected with no way to restore the original state.

Recommendations

Allow `address(0)` in `setRecipients()` to clear the mapping (restore direct claims) or add a dedicated recipient-removal function.

[I-01] The incorrect RPC URL equality check in the deployment script for `HplToken.sol` can misdetect the network

Description

The deployment script for `HplToken.sol` smart contract determines mainnet vs testnet using an exact string comparison of the RPC URL:

```
const IS_MAINNET = hre.network.config.url ==
"https://rpc.hyperliquid.xyz/evm";
```

Any variation in the configured RPC (provider change, trailing slash, query params, alias endpoint, etc,) flips detection. This can select the wrong `hyperCoreIndex`, which directly determines the token mint recipient, leading to full supply minted to an unintended address. Note: this is more deployment-time operational risk rather than an on-chain vulnerability.

Recommendations

Detect network via `chainId` or Hardhat `network.name` instead of RPC URL string comparison.

[I-02] No getter function for stored `hyperCoreSpotDeployer` in `HplToken.sol` contract

Description

`hyperCoreSpotDeployer` is stored in a custom storage slot but has no getter function. The value can only be read off-chain via `eth_getStorageAt`:

```
eth_getStorageAt(proxy,
0x8a35acfbc15ff81a39ae7d344fd709f28e8600b4aa8c65c6b64bfe7fe36bd19b)
```

This prevents other contracts from verifying the deployer on-chain and in general it is a good practice to have getter for these type of variables. Otherwise, the current approach is fine for off-chain verification.

Recommendations

If on-chain readability is needed, add a getter:

```
function getHyperCoreSpotDeployer() external view returns (address
deployer) {
    bytes32 slot = keccak256("HyperCore deployer");
    assembly {
        deployer := sload(slot)
    }
}
```

[I-03] Missing zero-address/zero-bytes32 validation in `MerkleDistributorForContracts.sol` constructor

Description

`token_` and `merkleRoot_` params are immutable with no validation:

```
constructor(address token_, bytes32 merkleRoot_, uint256 endTime_) {
    if (endTime_ <= block.timestamp) revert EndTimeInPast();
    token = token_;        // no check
    merkleRoot = merkleRoot_;  // no check
}
```

Deploying with `address(0)` or `bytes32(0)` creates a permanently broken contract. Claims would always fail and any tokens sent become stuck until `endTime`.

Recommendations

Add validation:

```
if (token_ == address(0)) revert InvalidToken();
if (merkleRoot_ == bytes32(0)) revert InvalidMerkleRoot();
```

[I-04] `MerkleDistributorForContracts.sol#claim()` function wrongly emits recipient in event when no redirect is configured

Description

When no recipient redirect is set (`recipients[account] == address(0)`), tokens are always transferred to `account`. However, the `recipient` function argument is not validated in this path and is emitted directly in the `ClaimedTo` event.

This allows any caller to can log any address they want in the event even though funds went elsewhere. This breaks event-based tracking for indexers and dashboards.

```
    function claim(uint256 index, address account, uint256 amount,
bytes32[] calldata merkleProof, address recipient) public virtual override
{
        // ... code ...

        // Send tokens to recipient instead of account
        if (recipients[account] != address(0)){
            if (recipient != recipients[account]) revert
InvalidRecipient(); //double check that recipient set by owner is the one
claimer wants to use
            IERC20(token).safeTransfer(recipients[account], amount);
        } else {
            IERC20(token).safeTransfer(account, amount); // <-- funds go
to the account
        }

        emit ClaimedTo(index, account, amount, recipient); // <-- emit
ClaimedTo event
    }
```

Recommendations

Emit the actual destination address (account in this case).

[I-05] Lack of two step ownership transfer implementation

Description

The MerkleDistributorForContracts contract implements Ownable, which implements single step ownership transfer pattern. In the event of ownership transfer to incorrect account, access to all functions protected by `onlyOwner` modifier is permanently lost.

```
contract MerkleDistributorForContracts is IMerkleDistributor, Ownable {
```

Recommendations

It is recommended to implement Ownable2Step instead.

[I-06] OpenZeppelin library version 4.7.0 has known vulnerabilities

Description

The protocol implemented within the `hyperlendx/merkle-distributor` repository makes use of OpenZepplin library version 4.7.0 that has known vulnerabilities, including the one related to `multiproofs` verification within the MerkleProof implementation. Usage of insecure libraries increases likelihood of exploit unknown vulnerabilities, that remain in the old code.

Recommendations

It is recommended to upgrade the code to the newest possible, without known vulnerabilities.

[I-07] HplToken makes use of Transparent Proxy

Description

The HplToken makes use of Transparent Proxy, however, UUPS proxy is considered superior due to following reasons:

- It offers flexibility to remove upgradeability.
- It is more gas-efficient than the Transparent Proxy pattern.

Recommendations

It is recommended to consider switching to UUPS proxy pattern.

[I-08] Arbitrary recipient can be set by claimer

Description

Whenever a claiming user is an account without recipient set within the recipients collection the arbitrary value can be set for `recipient` input parameter. Then, this value will be emitted within the `ClaimedTo` event. While this scenario has no impact on smart contract state, it may have impact on off-chain processing.

```
   function claim(uint256 index, address account, uint256 amount,
bytes32[] calldata merkleProof, address recipient)
        public
        virtual
        override
    {
        if (block.timestamp > endTime) revert ClaimWindowFinished();
        if (isClaimed(index)) revert AlreadyClaimed();
        if (msg.sender != account && msg.sender != recipients[account])
revert ClaimerNotAuthorized();

        // Verify the merkle proof.
        bytes32 node = keccak256(abi.encodePacked(index, account,
amount));
        if (!MerkleProof.verify(merkleProof, merkleRoot, node)) revert
InvalidProof();

        // Mark it claimed and send the token.
        _setClaimed(index);

        // Send tokens to recipient instead of account
        if (recipients[account] != address(0)){
            if (recipient != recipients[account]) revert
InvalidRecipient(); //double check that recipient set by owner is the one
claimer wants to use
            IERC20(token).safeTransfer(recipients[account], amount);
        } else {
            IERC20(token).safeTransfer(account, amount);
        }

        emit ClaimedTo(index, account, amount, recipient);
    }
```

Recommendations

It is recommended to set override `recipient` to `account` in certain condition.

[I-09] `block.timestamp` equal to `endTime` instance is not handled

Description

```
    function claim(uint256 index, address account, uint256 amount,
bytes32[] calldata merkleProof, address recipient)
        public
        virtual
        override
    {
        if (block.timestamp > endTime) revert ClaimWindowFinished();
...
```

```
    function withdraw() external onlyOwner {
        if (block.timestamp < endTime) revert NoWithdrawDuringClaim();
        IERC20(token).safeTransfer(msg.sender,
IERC20(token).balanceOf(address(this)));
    }
```

Recommendations

It is recommended to update the assertions within one of the aforementioned function to either >= or <= to handle all possible instnaces.

[I-10] HplToken is not compliant with EIP-7201

Description

The HplToken makes use of a namespaced storage slot to store the value of `hyperCoreSpotDeployer`. However, it is not compliant with the EIP-7201 standard. The proposed formula is as follows:

```
keccak256(keccak256(namespace) - 1) & ~0xff
```

Additionally, the slot's namespace can be pre-calculated in advance to save some Gas.

```
        // store deployer address for customStorageSlot verification
        require(hyperCoreSpotDeployer != address(0),
"hyperCoreSpotDeployer == address(0)");
        bytes32 verificationStorageSlot = keccak256("HyperCore deployer");
        assembly {
            sstore(verificationStorageSlot, hyperCoreSpotDeployer)
        }
```

Recommendations

It is recommended to make contract implementation compliant with the EIP-7201 standard.