Ledger Dev Cookbook

A practical guide with self-contained, real-world recipes to help developers adopt and master Ledger integrations.

Chapters Overview

5. UX Best Practices with Ledger

- Designing wallet-friendly UX

- Prompting user before signing

- Error handling and rejection cases

6. Security Pitfalls to Avoid

- Common mistakes in blind signing

- Double-checking display fields

- Preventing malicious dApp interactions

Each chapter includes:

- Code snippet

- JSON payload

- Screenshot/GIF (Ledger display)

- CLI/test instructions

Ledger Dev Cookbook Chapter 1: Clear Signing for Beginners

Objective:

Help developers quickly understand and implement Clear Signing using Ledger devices, focusing on a basic ERC-20 token transfer.

---

1. Enable Clear Signing on Ledger Device

- Open Ledger Live
- Navigate to Settings > Experimental features
- Enable Developer Mode
- Ensure Ethereum app is updated to latest version
- Enable contract data and blind signing OFF (in Ethereum app settings on device)

---

2. What to Expect on Ledger Display

- Token name and amount
- Recipient address (truncated)
- Network name (Mainnet, Goerli, etc.)
- Gas fee and Accept confirmation

Sample display:

```
  > Transfer 1.00 USDC
  > To: 0x1234...BEEF
  > Network: Ethereum
```

> Max fees: 0.001 ETH
>
> Accept?

---

3. Signing an ERC-20 Transaction (Example)

Code (TypeScript using ethers.js):

```ts
import { ethers } from 'ethers';
import TransportWebUSB from '@ledgerhq/hw-transport-webusb';
import Eth from '@ledgerhq/hw-app-eth';

async function signERC20Transfer() {
  const transport = await TransportWebUSB.create();
  const eth = new Eth(transport);

  const provider = new ethers.JsonRpcProvider('https://rpc.ankr.com/eth');
  const tokenAddress = '0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48'; // USDC
  const recipient = '0xRecipientAddressHere';
  const amount = ethers.parseUnits('1.0', 6); // 1 USDC (6 decimals)

  const erc20Interface = new ethers.Interface([
    'function transfer(address to, uint256 amount) returns (bool)',
  ]);
  const data = erc20Interface.encodeFunctionData('transfer', [recipient, amount]);
  const fromAddress = '0xYourLedgerAddress'; // Needed for nonce
  const nonce = await provider.getTransactionCount(fromAddress);
  const gasPrice = await provider.getGasPrice();

  const tx = {
    to: tokenAddress,
    value: 0,
    data,
    nonce,
    gasLimit: 60000,
    gasPrice,
    chainId: 1,
  };

  const rawTxHex = ethers.serializeTransaction(tx).slice(2);
  const result = await eth.signTransaction("44'/60'/0'/0/0", rawTxHex);
  console.log('Ledger Signature:', result);

  await transport.close();
}
```

---

4. Testing the Flow (CLI)

- Load script in Node.js

- Confirm Ledger is connected via USB

- Accept transaction on device

---

5. Troubleshooting

- Ensure Ethereum app is open on device

- Check contract data is enabled

- Use `chrome://devices` to allow USB permissions

- Use a browser with WebUSB support (Chrome/Brave)

This chapter provides a complete starting point to use Clear Signing safely and correctly with Ledger.

Ledger Dev Cookbook Chapter 2: Using ERC-7730 in Real-World dApps

Objective:

Guide developers to use ERC-7730 Clear Signing standard in decentralized applications with real use cases like swaps and complex interactions.

---

1. What is ERC-7730?

- An Ethereum standard for structuring metadata to be displayed on Ledger devices during transaction signing.
- Helps developers expose key info (e.g., swap rate, tokens, slippage) in a human-readable format.
- Enables richer Clear Signing for dApps, improving UX and security.

Reference: https://eips.ethereum.org/EIPS/eip-7730

---

2. Use Case Example: Token Swap via Aggregator (e.g., 1inch or Paraswap)

JSON Metadata Example:

```json
{
  "context": {
    "type": "SWAP",
    "origin": "https://mydapp.exchange"
  },
  "display": [
    { "type": "title", "text": "Token Swap" },
    { "type": "amount", "text": "1.00 USDC" },
    { "type": "arrow" },
    { "type": "amount", "text": "0.00065 ETH" },
    { "type": "fee", "text": "0.3% Platform Fee" }
  ]
}
```

This metadata gets bundled and passed alongside the transaction so Ledger can parse and display it safely.

---

3. Code Integration Snippet (Simplified)

```ts
const metadata = {
  context: {
    type: "SWAP",
    origin: window.location.origin
  },
  display: [
    { type: "title", text: "Token Swap" },
    { type: "amount", text: "1.00 USDC" },
    { type: "arrow" },
    { type: "amount", text: "0.00065 ETH" },
    { type: "fee", text: "0.3% Platform Fee" }
  ]
};

const encodedMetadata = ethers.encodeBytes(metadata); // depends on signing lib
const tx = {
  to: swapRouter,
  data: swapCallData,
  value: 0,
  gasLimit: 120000,
  metadata: encodedMetadata // Ledger parses this for Clear Signing
};
```

---

4. Display on Ledger Device

Expected UI:

```
  > Token Swap
  > 1.00 USDC  0.00065 ETH
  > Platform Fee: 0.3%
  > Origin: mydapp.exchange
  > Accept?
```

---

5. Best Practices

- Always show token names, not addresses.
- Display expected rate, slippage or protocol fee.
- Include domain or origin in metadata.
- Validate metadata structure before signing.

---

6. Testing Tips

- Use Ledger Dev tools to emulate device display
- Debug metadata encoding before injecting to tx
- Use Goerli or Sepolia testnets with small transfers

This chapter gives developers a strong baseline to use ERC-7730 and enrich transaction UX with safe, user-verified signing flows.

Ledger Dev Cookbook Chapter 3: Permit and EIP-2612 Signing

Objective:

Explain how to implement and sign `permit()` calls using EIP-2612 with Ledger, enabling gasless token approvals via typed data signatures.

---

1. What is EIP-2612?

- An Ethereum standard that allows ERC-20 token approvals via signature.
- Avoids sending an `approve()` transaction (no gas cost for the user).
- Commonly used in DeFi dApps before swaps or staking.

Reference: https://eips.ethereum.org/EIPS/eip-2612

---

2. Use Case

A user signs a permit for a token like DAI or USDC to allow a dApp (spender) to transfer tokens on their behalf.

---

3. JSON Structure of a Permit (Typed Data)

```json
{
  "domain": {
    "name": "USD Coin",
    "version": "2",
    "chainId": 1,
    "verifyingContract": "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48"
  },
  "message": {
    "owner": "0xYourAddress",
    "spender": "0xSpenderAddress",
    "value": "1000000",
    "nonce": 0,
    "deadline": 9999999999
  },
  "types": {
    "Permit": [
      { "name": "owner", "type": "address" },
      { "name": "spender", "type": "address" },
      { "name": "value", "type": "uint256" },
      { "name": "nonce", "type": "uint256" },
      { "name": "deadline", "type": "uint256" }
    ]
  },
  "primaryType": "Permit"
}
```

---

4. Code Example with Ledger (ethers.js + @ledgerhq/hw-app-eth)

```ts
import { ethers } from 'ethers';
import TransportWebUSB from '@ledgerhq/hw-transport-webusb';
import Eth from '@ledgerhq/hw-app-eth';

const domain = {
  name: 'USD Coin',
  version: '2',
  chainId: 1,
  verifyingContract: '0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48'
};

const types = {
  Permit: [
    { name: 'owner', type: 'address' },
    { name: 'spender', type: 'address' },
    { name: 'value', type: 'uint256' },
    { name: 'nonce', type: 'uint256' },
    { name: 'deadline', type: 'uint256' }
  ]
};
```

```
const value = {
  owner: '0xYourAddress',
  spender: '0xSpenderAddress',
  value: ethers.parseUnits('1.0', 6),
  nonce: 0,
  deadline: Math.floor(Date.now() / 1000) + 3600
};

// Use ethers.js _signTypedData if not on Ledger,
// otherwise format and send to ledger.signEIP712()
```

---

5. Ledger Display Output (expected)

  > Permit USD Coin

  > Spender: 0xAbc...1234

  > Amount: 1.00

  > Expires: 1h

  > Accept?

---

6. Security Considerations

- Validate `deadline` and `spender` on-chain before using permit

- Prevent front-running or replays (check `nonce`)

- Display real values (not raw data) in UI and on Ledger

---

7. Tips

- Many ERC-20 tokens support EIP-2612 (DAI, USDC, AAVE)

- Use off-chain permit signing before swaps or liquidity adds

- Include simulation / test in dApp UI

This chapter helps developers adopt gasless approvals safely and build Ledger-friendly permit workflows.

Ledger Dev Cookbook Chapter 4: Hardhat + Ledger Setup

Objective:

Enable developers to integrate Ledger signing into Hardhat-based development and testing workflows.

---

1. Why Combine Hardhat and Ledger?

- Simulate real-world mainnet/testnet flows with physical signing
- Perform accurate gas estimation, payload validation, and contract tests
- Securely test production-ready signing scenarios

---

2. Requirements

- Node.js and npm
- Hardhat (`npm install --save-dev hardhat`)
- Ethers.js
- Ledger device (Nano X/S Plus)
- `@ledgerhq/hw-app-eth` and `@ledgerhq/hw-transport-node-hid` (for local USB)

```bash
npm install @ledgerhq/hw-app-eth @ledgerhq/hw-transport-node-hid ethers
```

---

3. Configure Hardhat

Use a custom signer that delegates signing to the Ledger device.

```ts
// ledgerSigner.ts
import TransportNodeHid from "@ledgerhq/hw-transport-node-hid";
import Eth from "@ledgerhq/hw-app-eth";
import { Signer, TypedDataSigner, providers } from "ethers";

export class LedgerSigner extends Signer implements TypedDataSigner {
  provider: providers.Provider;
  eth: Eth;

  constructor(provider: providers.Provider) {
    super();
    this.provider = provider;
  }

  async init() {
    const transport = await TransportNodeHid.create();
    this.eth = new Eth(transport);
    return this;
  }
  async getAddress(): Promise<string> {
    const { address } = await this.eth.getAddress("44'/60'/0'/0/0");
    return address;
  }

  async signTransaction(tx) {
    const rawTx = ethers.utils.serializeTransaction(tx).slice(2);
    const sig = await this.eth.signTransaction("44'/60'/0'/0/0", rawTx);
    return ethers.utils.serializeTransaction(tx, {
      v: parseInt(sig.v, 16),
      r: "0x" + sig.r,
      s: "0x" + sig.s,
    });
  }

  async signTypedData() {
    throw new Error("Use a library to support EIP-712 with Ledger");
  }

  connect(provider) {
    return new LedgerSigner(provider);
  }
}
```

---

4. Using Ledger in Hardhat Test

```ts
import { LedgerSigner } from "./ledgerSigner";

task("transfer", "Send a token using Ledger").setAction(async (_, hre) => {
  const provider = hre.ethers.provider;
  const ledger = await new LedgerSigner(provider).init();

  const token = await hre.ethers.getContractAt("IERC20", "0xTokenAddress", ledger);
  const tx = await token.transfer("0xRecipient", ethers.utils.parseUnits("1.0", 18));
  console.log("Sent:", tx.hash);
});
```

---

5. Common Issues & Tips

- Ensure Ledger is unlocked and Ethereum app is open
- On Linux/macOS, add proper udev rules or USB permissions
- Disable browser apps that may block USB HID (e.g., Ledger Live)
- Use `--network goerli` for testing

---

6. Additional Ideas

- Use Ledger signer in `hre.ethers.getSigners()` override
- Test on localhost fork of mainnet for real token behavior
- Combine with permit() signing or ERC-7730 display tests

This chapter helps integrate secure hardware signing into developer tooling using Hardhat and Ledger devices.

Ledger Dev Cookbook Chapter 5: UX Best Practices with Ledger

Objective:

Guide developers to build user interfaces and flows that complement Ledger devices and promote safe, confident signing experiences.

---

1. Why UX Matters with Ledger

- Signing on a hardware wallet is physical and sequential
- Users need clarity, not just correctness
- A bad UI can lead to blind signing or mistrust
- A good UX reinforces trust and confidence in the app

---

2. Design Principles

Show key details BEFORE the user clicks Sign

Match app values to what is displayed on the Ledger

Warn the user if fields will not appear on Ledger

Display token names, amounts, and destination addresses

Provide a preview and Confirm on device step

---

3. UX Checklist for Ledger Integration

- [ ] Show the full summary of the transaction: token, amount, receiver
- [ ] Indicate when the user needs to look at their device
- [ ] Highlight if a signature includes contract interaction
- [ ] Use plain language to explain what will happen
- [ ] Warn when data may not be verifiable (e.g. blind signing)

---

4. Good UX Example: Swap Interface

App preview:

```
 You are about to swap:

   1.00 USDC  0.00065 ETH

 This action will be signed securely with your Ledger device.

 Please confirm on your Ledger:
   - Token in: USDC
   - Token out: ETH
   - Rate: 1534.6
   - Max slippage: 0.5%
```

Device shows:

> Token Swap
>
> 1.00 USDC 0.00065 ETH
>
> Platform Fee: 0.3%
>
> Accept?

---

5. Handling Errors Gracefully

- Device disconnected Ledger not found. Please reconnect.
- Wrong app open Please open Ethereum app on your Ledger.
- Data mismatch Transaction could not be signed. Please try again.

Always provide a retry option and context-specific tips.

---

6. Educating the User

Add tooltips or links:

What is Clear Signing?

Why you should always verify amounts and addresses on your device.

This builds trust and empowers even non-technical users to understand security.

---

7. Dark Patterns to Avoid

Auto-signing without a Ledger prompt

Hiding what is being signed

Mixing fields visually (e.g. recipient/address/fee in the same line)

Using truncated or base64-encoded fields with no explanation

---

8. Bonus: Accessibility

- Keyboard navigation and focus handling for Sign steps
- Screen reader support for blind users
- Haptic/audio prompts to assist the check device moment

Great UX encourages safe, confident Ledger signing. Poor UX risks user funds.

This chapter gives the best practices to align frontend with secure hardware flows.

Ledger Dev Cookbook Chapter 6: Security Pitfalls to Avoid

Objective:

Help developers avoid common security mistakes when integrating Ledger hardware wallets into dApps and workflows.

---

1. Blind Signing

 Problem: Signing hex or arbitrary payloads without display

 Solution: Use Clear Signing with ERC-7730 and typed data (EIP-712)

Blind signing increases the risk of phishing and malicious contracts. Always favor human-readable display.

---

2. Mismatched UI and Ledger Display

 Problem: Displaying different info on app and Ledger

 Solution: Make sure all critical values (amount, token, recipient) match exactly

This mismatch can confuse users or lead to signing unintended transactions.

---

3. Not Validating Inputs

Problem: Using unchecked inputs from frontend to build tx

Solution: Sanitize and validate all addresses, token decimals, and amounts

Example: A typo in the token address may send funds to an incorrect contract.

---

4. Ignoring Signature Nonce or Deadline

Problem: Reusing permits or allowing replay

Solution: Always verify nonce and enforce deadline limits

EIP-2612 permits must be used only once and within their allowed time window.

---

5. Skipping Origin Checks

Problem: Using metadata without checking dApp origin

Solution: Bind `origin` field in ERC-7730 and validate before signing

This prevents dApps from spoofing others' metadata or phishing users.

---

6. Hardcoding Derivation Paths

Problem: Using fixed paths like "44'/60'/0'/0/0" blindly

Solution: Let users choose their account, or scan addresses dynamically

Different wallets may use different paths. Support account selection.

---

7. Not Handling Errors Securely

Problem: Logging sensitive payloads or crashing on user rejection

Solution: Gracefully handle `.reject()` and never log private data

Always provide feedback and safe fallbacks without leaking user info.

---

8. Not Using Contract Whitelisting (Optional)

Solution: For high-security use cases, maintain a registry of trusted contract addresses to limit interaction scope.

This can prevent rogue contracts from tricking the user into unsafe approvals or transfers.

Security is not only backendit includes UX, signing flow, validation, and recovery paths.

Ledger-based dApps must be safe by design, not just by code.