Extending Tokens with ERC20Votes

Today's lesson focuses on extending tokens, specifically using the **ERC20Votes** extension. This extension offers a way to incorporate voting mechanisms into tokens, moving beyond simple token balances to track voting power over time.

Why ERC20Votes?

- **Ballot System Analogy:** Similar to the ballot system, ERC20Votes allows token holders to cast votes. Instead of a single chairperson granting voting rights, token balances or voting power determine the weight of each vote.
- **Extensibility:** OpenZeppelin offers various token formats, upgrade patterns, and extensions (e.g., pausable, mintable, burnable). ERC20Votes serves as an example of how to apply these extensions.
- **Not the Only Way:** While ERC20Votes is highlighted, it's crucial to understand it as an example of how to implement extensions for tokens.

Practical Application: Tokenized Ballot

The goal is to build a tokenized ballot system, similar to last week's ballot.sol, but using a tokenized approach. This involves:

- **Token Distribution:** Implement scripts for distributing voting tokens.
- **Token Mechanics:**
    - Minting tokens
    - Delegating tokens
    - Casting votes
    - Checking voting power
- **Reusability:** You can reuse some scripts from the previous ballot implementation.

ERC20Votes Explained

The ERC20Votes extension maintains a history of **checkpoints** for each account's voting power. This means each address has a voting power value that can change over time based on these checkpoints.

Key Features:

- **Delegation:** Users can delegate their voting power to others, similar to the previous ballot system.
    - Direct delegation via a `delegate` function.
    - Delegation by signature using a `permit` function.
- **Public Access:** Voting power can be accessed publicly using functions like `getVotingPower` (current) and `getPastVotes` (historical).

Core Functions

- `getVotes`
- `getPastVotes`

Design Considerations

- **Gas Efficiency:** To reduce gas costs, voting power isn't automatically tied to token balance changes.
- **Activation via Delegation:** Voting power tracking is activated when a user delegates tokens to themselves. This creates the initial checkpoint.

Technical Details

- **Max Supply:** The maximum token supply is reduced to 2^{208} (from the usual $2^{256} - 1$ for standard ERC20 tokens).

Questions and Answers

- **Token Ownership vs. Voting Power:** Owning a token doesn't automatically grant voting power. Delegation is required to activate voting power.

Voting Power and Delegation

Delegation Scenario

Imagine a boot camp scenario where you only need voting power at the end. Instead of paying gas fees repeatedly to increase your voting power, you would:

1. Collect all the money during the boot camp.
2. Delegate to yourself before the final voting ballot.
3. Activate a checkpoint to use your voting power.

Extension on ERC20s

- This voting system is an extension on **ERC20s**, meaning it has **fungible votes**.
- You can vote with a fraction of a token (e.g., 0.001) instead of a whole unit.
- It inherits and modifies aspects of ERC20 tokens.

Smart Contract Documentation

Importance of Documentation

Smart contract documentation can sometimes be unclear. Even with extensive documentation, understanding a smart contract can be challenging.

OpenZeppelin's Documentation

OpenZeppelin's documentation is among the best, but it can still be concise.

Understanding Through Testing

The best way to fully grasp a smart contract is by:

1. Running it on a local machine.
2. Testing its functionalities.
3. Observing how it behaves and if it breaks.

ERC20 Permit

Overview

ERC20 Permit is an extension that uses signatures to emit approvals off-chain without paying gas fees. This extension is utilized by ERC20 Votes.

Process

1. Normally, when interacting with an ERC20 token, you need to **approve** the contract first before it can move your tokens.
2. With ERC20 Permit, you sign a message **off-chain** with your wallet, which is then sent to the contract.
3. This method allows you to change the allowance of an account by presenting a signed message.

Benefits

- **No Gas Fees**: Users don't need to pay gas for approvals.
- **Flexibility**: The signature can be sent via various channels like email or Telegram.

Security

If the signature is leaked, it can lead to problems, such as theft of funds. It's crucial to keep the signature secure and send it only to the intended recipient.

Use Case: Airdrops

When receiving an airdrop of a voting token, you might not have the network token to pay for gas. ERC20 Permit allows you to use the voting token without needing the network token.

OpenZeppelin Wizard

Purpose

The OpenZeppelin Wizard is a tool to build smart contracts. It allows you to create various types of smart contracts (e.g., stablecoins, reward assets, DAOs).

Building a Voting Smart Contract

The wizard can be used to build a voting smart contract that keeps track of historical balances for on-chain governance.

Configuration Options

- Block Number: Use block numbers as the default version for voting.
- Mintable with Roles: Configure the contract to be mintable with access control roles.
- Access Control: Include access control for managing permissions.

Inherited Contracts

- ERC20: Inherits the standard ERC20 implementation.
- ERC20 Permit: Applies the ERC20 Permit extension.
- ERC20 Votes: Applies the ERC20 Votes extension.

Overrides

Solidity requires explicit definition of overrides when multiple contracts are applied on top of each other, especially when one contract overrides the behavior of another.

Example Functions

- `update`: A function needed to solve overrides in ERC20 and ERC20 Votes.
- `_nonces`: Solves the clash between ERC20 Permit and nonces.

Inheritance

When applying multiple contracts on top of each other, the behavior from one contract might override another. Solidity requires explicit definition of these overrides. If you don't define these overrides, the code might not work.

Project Setup & Initialization

Creating a New Project Structure

To start a new project for testing or designing smart contracts:

1. Copy an existing project folder.
2. Delete unnecessary folders and files like `arax`, `cache`, `ignition`, `scripts`, and `test`.
3. Create new folders for `contracts` and `scripts`.
4. Run `npm install` to ensure all dependencies are installed.

   This process should only take a couple of minutes, enabling quick setup for testing smart contracts, swaps, or DEX operations.

Dependency Management with OpenZeppelin Contracts

- When installing **OpenZeppelin Contracts** using `npm install`, all its dependencies are automatically installed.
- Ensure `openzeppelin-contracts` is listed in your `package.json` file.

Smart Contract Implementation

Using the OpenZeppelin Wizard

The **OpenZeppelin Wizard** can be used to generate smart contracts by selecting desired features:

- Go to the OpenZeppelin Wizard to build your smart contract by checking the boxes for the features you need.
- The wizard is akin to building a Subway sandwich: you select ingredients (features) like extra bacon, double cheese, etc., and the final taste (functionality) depends on your choices.

Resolving Solidity Compiler Errors

When working with derived contracts, you might encounter errors related to function overriding:

- If Solidity complains about a derived contract and function overrides, add the necessary override to appease the compiler.
- This doesn't change the smart contract's behavior but satisfies Solidity's requirements.
- The compiler is simply checking if you know what you are doing. By adding the overrides, you are telling the compiler that you are aware of the function overrides and that you want to compile it anyway.

Testing Smart Contracts

Creating Test Scripts

Instead of diving deep into complex smart contract code, use test scripts to understand functionality:

1. Create a new script file (e.g., `testVoteToken.ts`).
2. Implement a test script to interact with the smart contract.
3. This approach facilitates easier understanding and validation of contract behavior.

Replicating Script Structure

Utilize a consistent structure for test scripts:

- Copy the structure from previous lessons (e.g., Lesson 9) including the main function, public client, and getWalletClients.
- Include a `try-catch` block for error handling.
- Deploy the contract within the script and log the deployment address.

```
async function main() {
  try {
    // Your code here
  } catch (error) {
    console.error(error);
  }
}
```##  Smart Contract Testing

Starting Base for Smart Contract Tests

*   When writing smart contracts, start with a base script that includes a
*   This base allows for deploying smart contracts, sending transactions,

Deploying a Smart Contract
*   The lecture demonstrates the process of deploying a smart contract to

Adjusting Smart Contracts

*   Initially, the smart contract required parameters during deployment, s
*   The contract was modified to use `msg.sender` as the default admin, si

Understanding Design Changes

*   A design change was implemented where instead of assuming `msg.sender`

Importance of Backticks for String Formatting

*   When formatting strings, use **backticks** instead of normal quotes to

□ Minting Tokens

*   To mint tokens, a mint transaction is created.

    ```js
    // Example: Minting tokens to account one
    accountOne.mint(accountOneAddress, mintValue);
    ```
*   `mintValue` can be defined as a constant.

    ```js
    const mintValue = 10;
    ```

Units and Decimals

*   It's crucial to understand how units and decimals work when minting to
*   Using just `10` may only mint 10 decimal units instead of 10 entire to
*   Use `parseEther` to convert from a string to a BigInt to handle the de

    ```js
    parseEther("10"); // Mints 10 tokens with all 18 decimal places
    ```
*   `formatEther` can be used to display the tokens later.

```
### ERC20 Token Behavior

*   The token behaves like a normal **ERC20 token**, allowing for:
    *   Total supply tracking
    *   Transfers
    *   Approvals

###  Voting Power and Delegation

*   A newly minted token does not automatically grant voting power.
*   Even with 10 tokens, the voting power might initially be zero.
*   To activate voting power, **self-delegation** is required.

###  Gas Savings

*   The need for self-delegation is a design choice to make transfers chea

### Delegation Transaction

*   To delegate voting power, a **delegate transaction** must be created.
*   This transaction involves calling the `delegate` function, which comes
##  Self-Delegation and Voting Power
Self-delegating involves an account delegating its voting power to itself.

### Demonstration of Self-Delegation
Before self-delegating, an account may have zero voting power. After self-

##  Token Transfers and Voting Power
### Transfer Scenario: Alice and Bob
Consider Alice and Bob, where Alice (account one) has 10 tokens and transf

### Voting Power After Transfer
When Alice transfers tokens, her voting power is instantly updated. She do

### Example Scenario Breakdown
1.  Initial state:
    *   Alice has 10 tokens.
    *   Total supply is 10.
    *   Total voting power is zero.
2.  Alice activates her voting power:
    *   Total supply remains 10.
    *   Total voting power is now 10.
3.  Alice transfers 5 tokens to Bob:
    *   Total supply remains 10.
    *   Total voting power becomes 5 (Alice's remaining tokens).

##  Self-Delegation for Receivers
Bob needs to self-delegate to activate his voting power. A delegate transf

##  Historical Voting Power

### Accessing Past Votes
It's possible to loop through past blocks to determine the past votes for

### Big Number Context
"BN" refers to "big number," used to handle large numbers in calculations.
```

```
    >Big Number (BN) - Used to handle large numbers in calculations.

### Practical Use of Historical Values
Historical voting power values can be used to create snapshots for voting

### Example: Ice Cream Flavor Vote
Imagine a scenario where a vote is announced suddenly (e.g., voting for an

##  Questions and Clarifications

### Clarification on Account Two's Voting Power
After receiving a transfer, account two (Bob) still has zero voting power

## Voting Power and Block History

After receiving tokens, the account had zero units of voting power. After

You can also use **logging** to create events that you can go through to m

## Transfer Function Parameters

The parameter for the transfer function should be one address and one numb

## Accessing Past Votes

You cannot get past votes on the current block. This is why the last trans

There is a way to force mining transactions, which will be covered in futu

## Constants vs. Variables in Test Files

Defining variables as **constants** at the top of the script allows for ea

> Fuzzing: a smart contract security technique that involves trying differ

It is good practice to extract settings in your code to have one place to

### Constants and Debugging

Declaring multiple transactions as constants helps with debugging. Each co

Using the same variable for everything can cause you to lose the history o

## Weekend Project: Tokenized Ballot

The weekend project involves creating a **tokenized ballot**, similar to p

### Key Differences in the Tokenized Ballot
*   No struct of voters.
*   No function delegate.
*   The `vote` function takes two parameters: `proposal` and `amount`.

### Token Contract Integration

The token contract handles most of the voting power and delegation aspects

### Structs and Functions
```

```
The struct of proposals is still needed. However, the voting power of some

### Smart Contract Code Example

Create a smart contract called `TokenizedBallot.sol` inside the contracts

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract TokenizedBallot {
    struct Proposal {
        string name;
        uint voteCount;
    }

    address public tokenContractAddress;
    Proposal[] public proposals;

    constructor(address _tokenContractAddress, string[] memory _proposalNa
        tokenContractAddress = _tokenContractAddress;
        proposals = new Proposal[](_proposalNames.length);
        for (uint i = 0; i < _proposalNames.length; i++) {
            proposals[i].name = _proposalNames[i];
        }
    }

    function vote(uint proposal, uint amount) public {
        // Implementation
    }
}
```

Voting Powers Inside Token Contract

The voting powers are located inside the token contract itself. A ballot smart contract holds the votes and consumes them.

Weekend Project Overview

The weekend project involves completing contracts as a group. The goal is to understand the voting mechanisms by working through the solution together and creating scripts.

Project Settings
```

- **Proposals array:** Similar to the previous ballot, when deploying the contract, an array of proposals is available.
- **Token contract attachment:** The ballot is attached to a token contract, which determines the voting powers for each person.
- **Target block number:** A cutoff date is set to create a cliff period between the snapshot date of voting powers and the actual voting time, preventing vote buying.

Addressing Voting Power

Accessing Voting Power

To see who has voting power from the token contract, the contract is called using `getVotes` to view a person's voting power. There isn't a list that keeps track of who can vote. The focus is on increasing the vote count of a proposal by a certain amount.

Checks Before Voting

Before increasing the vote count, the voting power of the account calling the function is verified.

`getVotingPower` Function

```
function getVotingPower(address voter) public view returns (uint256) {
    // Implementation details
}
```

To implement this, an interface of the contract is used to get past votes from the voter, passing the target block number.

Remaining Voting Power

Calculating Remaining Voting Power

The remaining voting power is calculated by subtracting the amount a person has already spent from their total voting power.

remainingVotingPower = totalVotingPower - amountSpent

Tracking Spent Amounts

A mapping is used to track the amount of votes each address has spent in the ballot.

```
mapping(address => uint256) public votePowerSpent;
```

Each time a person votes, the `votePowerSpent` for that person increases by the amount they voted. The remaining voting power is then adjusted accordingly.

Weekend Project Tasks

- Complete the contract.
- Write the tests to test the contract.
- Create scripts for:
    - Deploying
    - Minting voting tokens
    - Delegating voting power
    - Deploying the ballot after self-delegation (with a locked target number)
    - Casting votes
    - Checking voting power
    - Checking winning proposals

Questions and Feedback

- GitHub issues can be opened for questions or tests to receive feedback.
- Issues are answered daily or every couple of days.