Smart Contract Deep Dive

Function Calling Flow

In the EVM (Ethereum Virtual Machine), when calling a function in a smart contract, you don't pass the entire function name string. Instead, you pass only **four bytes**, known as the **function selector**.

When a function is called in a smart contract, certain attributes must be defined:

- **Visibility**: Where the function can be called from
- **Mutability**: Whether the function can modify the state of the contract
- **Return**: What the function returns

All of these build the function selector.

Here's how it works:

1. **Method ID**: Each function is identified by a **method ID**, which is the first four bytes of the Keccak-256 hash of the function's signature.
2. **Function Signature**: The function signature includes the function name and the data types of its parameters.
3. **Function Selector**: The EVM uses this four-byte selector to identify and call the correct function.

Decoding Function Calls

Let's analyze a transaction to see the function selector in action:

- When you make a transaction, the input data starts with the four-byte function selector.
- For example, calling `setText` might look like this in the transaction input: `5d3a1f9d`.
- This `5d3a1f9d` is the function selector, which the EVM uses to identify the `setText` function.

```
"Decoding this input, I know that I'm calling the function `setText` to be the
```

The function selector is calculated by hashing the function name and the types of the parameters and then picking the first four bytes of the hash. Changing the function name or the types of the parameters will change the signature.

Interfaces and Function Selectors

An **interface** defines the functions that a contract should implement, but it doesn't include the implementation details. Interfaces work with function selectors as well:

1. **Define Interface**: Create an interface with function definitions, including names, parameters, and return types.
2. **Attach Interface**: When attaching an interface to a smart contract, the compiler checks if the contract implements all the functions defined in the interface.

Example: Interface Creation

Consider a smart contract with functions like `helloWorld` and `setText`. Let's create an interface for it.

```
interface IHelloWorld {
    function helloWorld() external;
    function setText(string calldata _newText) external;
}
```

In this interface:

- `helloWorld` and `setText` are defined with their respective signatures and marked as `external`.
- If you try to attach this interface to a smart contract that doesn't implement these functions, the compiler will throw an error.

Interface Compliance

When you attach an interface to a smart contract, the compiler should check if the contract implements all the functions defined in the interface. In some cases, there might be no strict conformity check when attaching an interface to a smart contract. This is because the EVM doesn't inherently know the functions defined in the interface.

Function Signatures and Calling Smart Contracts

When interacting with smart contracts, function signatures play a crucial role. Even without knowing the function names, you can interact with a smart contract by sending specific bytes corresponding to the function signature.

- **Function Signature**: The first four bytes of the Keccak-256 hash of the function's signature.
- A function signature is the unique identifier that is used to call functions within a smart contract.
- When calling a function, the function signature is sent as part of the transaction data.

Calling Functions by Signature

Any function signature in a smart contract can be called to check its response. The name of the function is not required.

1. Send a transaction with the function signature.
2. If the smart contract has a function corresponding to the signature, it executes.
3. If no function exists, the call reverts.

For example, to call a function, you need to pass the correct byte data. If the data is incorrect, the transaction will fail.

Replay Attacks

A replay attack involves reusing transaction data from past transactions.

- Attackers can observe past transactions and extract the data sent to a smart contract.
- The attacker can then replay the call by sending the same data or tweak the input data.

Hash Collisions and the Compiler

- **Hash Collision**: When two different methods have the same hash.

The compiler handles hash collisions. If two methods have the same hash, the compiler prepends one or two characters to differentiate them. Hash collisions become relevant in proxy patterns, which are covered in later lessons.

External Calls

An external call occurs when one smart contract calls another, without needing inheritance. It involves one entity calling another entity.

Greetings Invoker Example

Consider a contract called `GreetingsInvoker` that calls another contract. The `GreetingsInvoker` contract doesn't perform actions on its own but invokes functions in another contract.

```
contract GreetingInvoker {
    function invokeGreeting(address target) public view returns (string me
        // Implementation to call another contract
    }
}
```

In this example, the `invokeGreeting` function calls a function in another contract at the specified address.

Low-Level Calls vs. Type Casting

There are two ways to make external calls:

1. **Low-Level Call**:
    - Involves using `encodeWithSignature` to encode the function call.
    - More complex and less readable.

```
(bool success, bytes memory data) = target.call(abi.encodeWithSignatu
```

2. **Type Casting**:
    - Casting the target address to an interface.
    - More readable and maintainable.

```
interface IHello {
    function helloGreeting() external view returns (string memory);
}

function invokeGreeting(address target) public view returns (string m
    IHello hello = IHello(target);
    return hello.helloGreeting();
}
```

Demonstrating External Calls

1. Deploy two smart contracts: `HelloWorld` and `GreetingsInvoker`.
2. Set the text in `HelloWorld` to a specific value (e.g., "something else").
3. Call `invokeGreeting` in `GreetingsInvoker`, passing the address of `HelloWorld`.
4. `GreetingsInvoker` calls the `helloGreeting` function in `HelloWorld` and returns the stored text.

If the target address does not implement the required function (e.g., an Externally Owned Account (EOA) or a contract without the function), the transaction will fail.

Interfaces

- Every contract has an interface, but the interface does not contain the implementation.
- If the contract itself is unavailable, the interface can be created or low-level calls with the correct bytes can be used.

Changing State in Another Contract

Smart contracts can make state-changing calls in other smart contracts. Contract A can call Contract B, which can call Contract C, and so on.

External Contract Interactions

One smart contract can interact with another in the same way an Externally Owned Account (EOA) interacts with a smart contract.

How External Calls Work

The process involves:

1. **Finding the four-byte function selector:** This can be obtained from the contract's code or past transactions.
2. **Passing the function selector and arguments:** Use this information to interact with the desired function in the external contract.

Questions and Answers

Parameter Requirement

When calling a function in another contract, you must provide all the parameters that the function expects. For example, if a function `setText(string newText)` is being called, you need to provide a `newText` parameter.

Security Concerns

Any public method in a smart contract can be executed externally. It's possible for someone to attach an interface to your smart contract and call its functions, even if you deployed the contract.

Security Risks vs. Features

While anyone can call any public function in a smart contract, it's not inherently a security risk but a feature of smart contracts. However, it can lead to hacks if not properly controlled.

Limiting Function Calls: Access Control

To restrict function calls to specific individuals, you can implement Access Control.

Playing with Smart Contracts

Treat deployed smart contracts as a playground for learning. Experiment by:

- Trying different inputs (e.g., numbers instead of strings).
- Reversing actions (e.g., changing text back to its original value).
- Keeping a history of changes.

Contract Addresses

To interact with a deployed contract, you need its address. This address can be shared so others can interact with the contract.

Receive and Fallback Functions

Smart contracts have special functions that define how they handle receiving Ether:

- Receive Ether Function
- Fallback Function

Receive Ether Function

> A function that enables a smart contract to receive Ether. By default, smart contracts cannot receive Ether without a receive function or a payable fallback function. When a contract receives Ether with empty call data, the receive function is executed.

If you try to send Ether to a contract that doesn't have a receive function, the transaction will revert. Here's an example of a receive function in Solidity:

```
receive() external payable {
    // Logic to handle receiving Ether
}
```

Fallback Function

The lecture mentions a fallback function but does not go into detail regarding what it is.

Invoking Non-Existent Functions

If you try to call a function that doesn't exist in a contract, the transaction will revert.

Fallback Functions in Solidity

Fallback Function Explained

A **fallback function** is a special function in a smart contract that is executed if none of the other functions match the given function signature. It acts as a "catch-all" case when an unrecognized function call is made to the contract.

- Can be used to handle calls to functions that don't exist in the contract.
- Commonly used in proxy patterns.
- Can be tricky and cause security vulnerabilities if not handled correctly.

Example Fallback Implementation

Here's an example of a fallback function in Solidity:

```
pragma solidity ^0.8.0;

contract Hello {
    string public text;

    constructor() {
        text = "Hello World";
    }

    function hello() public view returns (string memory) {
        return text;
    }

    function setText(string memory newText) public {
        text = newText;
    }

    function initialText() public {
        text = "Hello World";
    }

    fallback() external {
        text = "Wrong";
    }
}
```

In this example, the fallback function sets the `text` state variable to "Wrong" if any function call doesn't match `hello`, `setText`, or `initialText`.

Important Considerations

- **Security:** Fallback functions can be exploited, especially with external calls.
- **Gas Costs:** Fallback functions should be gas-efficient.
- **Proxy Patterns:** Fallback functions are commonly used in proxy contracts to delegate calls.

Calling a Fallback Function

To call a fallback function, you need to call a function with a signature that doesn't match any of the explicitly defined functions in the contract. For example:

```
contract MyContract {
    string public data;

    fallback() external {
        data = "Fallback called!";
    }

    function normalFunction() public {
        data = "Normal function called!";
    }
}
```

To trigger the fallback, you could call the contract with a non-existent function selector like `0x00000001`.

Receiving Ether via Fallback

When a contract receives Ether without a specific function call, the fallback function is executed. To handle Ether transfers, you can make the fallback function payable:

```
fallback() external payable {
    if (msg.value > 0) {
        data = "Thank you for the Ether!";
    } else {
        data = "Fallback called without Ether.";
    }
}
```

Limitations

- You cannot directly target the fallback function from within the contract itself. It must be an external call.
- The fallback function is triggered only if the function selector doesn't match any other function in the contract.

Why Solidity's Grammar Matters

Solidity's function definition grammar is crucial because it directly relates to how function calls are handled in the Ethereum Virtual Machine (EVM). Understanding the grammar helps you:

- Appreciate the importance of declaring functions as `public` or `external`.
- Understand how the EVM uses function selectors to route calls.
- Grasp the significance of `view` and `pure` modifiers in terms of block storage and payment handling.

Solidity's design reflects the architecture of blockchains and the EVM.

Access Control in Smart Contracts

Introduction to Access Control

Access control restricts who can perform certain actions on a smart contract. It prevents unauthorized users from modifying the contract's state or executing sensitive functions.

Common Access Restriction Patterns

- **Owner-Based:** Only the contract's owner can execute certain functions.
- **Time-Based:** Actions are restricted based on time (e.g., creation time).
- **Role-Based:** Specific roles are assigned to addresses, and only those with the correct role can execute certain functions.

Implementing Owner-Based Access Control

To implement owner-based access control, you typically:

1. Store the owner's address in a state variable.
2. Check if the `msg.sender` (the caller's address) matches the owner's address before executing a restricted function.
3. Use `require` or `revert` to stop execution if the caller is not authorized.

Here's a simple example:

```
pragma solidity ^0.8.0;

contract AccessControl {
    address public owner;
    string public text;

    constructor() {
        owner = msg.sender;
        text = "Hello World";
    }

    function setText(string memory newText) public {
        require(msg.sender == owner, "Only the owner can set the text.");
        text = newText;
    }

    function getText() public view returns (string memory) {
        return text;
    }
}
```

In this example, only the contract owner can call the `setText` function.

Common Methods to Make Transactions Fail

To enforce access control, you can use these methods to revert a transaction if a condition is not met:

- **`require(condition, "Error message")`:** Checks a condition and reverts the transaction if the condition is false.
- **`assert(condition)`:** Checks a condition and reverts the transaction if the condition is false (primarily for internal errors and debugging).
- **`revert("Error message")`:** Unconditionally reverts the transaction with a custom error message.

Access Control in Smart Contracts

Requirements

Requirements are a way to ensure that certain conditions are met before a transaction can be executed. They can be used to control access to functions within a smart contract. For example, a requirement could specify that only the owner of the contract can call a particular function.

Requirements are typically expressed as conditional statements, such as:

```
require(msg.sender == owner, "Only the owner can call this function");
```

This requirement checks that the address sending the transaction (`msg.sender`) is equal to the contract owner's address. If the requirement is not met, the transaction is reverted.

Message Sender (`msg.sender`)

`msg.sender` is a global variable in Solidity that represents the **address of the account or smart contract that initiated the current transaction**. It is crucial for implementing access control and tracking interactions with smart contracts.

The `msg.sender` is not necessarily the original sender of the transaction if the transaction involves multiple smart contracts calling each other. In such cases, the `msg.sender` will be the address of the smart contract making the call.

Examples of Access Control

Here are some examples of access control mechanisms:

- Requiring that the person sending a transaction is equal to the contract's address.
- Requiring that the person sending a transaction is not equal to the contract's address.

Access Control Logic

Access control logic can be based on various factors, including:

- Time
- Hour
- Person
- String
- Payment

Demonstration of Access Control

Consider a smart contract with a `setText` function that only the owner of the contract should be able to call:

```
pragma solidity ^0.8.0;

contract HelloWorld {
    string public text;
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    function setText(string memory _text) public {
        require(msg.sender == owner, "Only the owner can set the text.");
        text = _text;
    }

    function hello() public view returns (string memory) {
        return text;
    }
}
```

In this example, the `require` statement ensures that only the owner of the contract can call the `setText` function. If someone else tries to call the function, the transaction will be reverted.

Determining the Deployer of a Smart Contract

There is no built-in variable in Solidity that directly stores the address of the account that deployed the smart contract. However, you can store the `msg.sender` in the constructor to capture the deployer's address.

```
pragma solidity ^0.8.0;

contract HelloWorld {
    address public owner;

    constructor() {
        owner = msg.sender;
    }
}
```

In this example, the constructor sets the `owner` state variable to the `msg.sender`, effectively storing the address of the deployer.

Visibility of State Variables

When a state variable is marked as `public`, the Solidity compiler automatically generates a **getter function** for that variable. This allows you to read the value of the state variable from outside the contract without having to write a separate getter function. Marking a state variable as public does not allow it to be changed.

```
pragma solidity ^0.8.0;

contract HelloWorld {
    address public owner; // Public state variable

    constructor() {
        owner = msg.sender;
    }
}
```

In this example, the compiler will automatically generate a function that allows you to read the value of the `owner` variable.

Creating Interfaces from ABIs

To create an interface from an Application Binary Interface (ABI), you need to define the function signatures based on the ABI information. The interface should include the function name, inputs, and outputs.

For example, if you have the ABI of an ERC20 token contract, you can create an interface like this:

```
interface IERC20 {
    function name() external view returns (string memory);
    function approve(address spender, uint256 amount) external returns (bo
}
```

Pure Functions

Pure functions are functions that do not read or modify the contract's state. They are often used for calculations or operations that depend only on the input parameters. Pure functions are more commonly found in standards like ERC721, especially when dealing with constants such as the URI of the smart contract.

Pure Functions

Pure functions do not rely on or modify the state of the blockchain. Examples include:

- String manipulation functions (e.g., slicing, getting length, copying, casting).

Access Control Implementation

Calling `setText`

The `setText` function can only be called by the owner.

`onlyOwner` Modifier

The `onlyOwner` modifier checks if `msg.sender` is the owner. If not, it reverts with the message "Caller is not the owner".

`transferOwnership` Function

The `transferOwnership` function changes the owner to a new address.

```
function transferOwnership(address newOwner) public onlyOwner {
    owner = newOwner;
}
```

Only the current owner can call this function.

Contract Deployment and Updates

- Deploying a contract creates a new instance at a new address on the blockchain.
- It is not possible to change a contract that has already been deployed.
- Each deployment, even with a minor change, results in a new contract instance.

Modifier Behavior

Modifier Structure

The _ character in a modifier indicates where the function execution should resume after the modifier code is executed.

Challenge: Modifier Placement

What happens if the _ character is placed at the top of the modifier?

```
modifier onlyOwner {
    _;
    require(msg.sender == owner, "Caller is not the owner");
}
```

Experiment

- Deploy the contract.
- Attempt to call `setText` with a non-owner account.
- Attempt to transfer ownership with a non-owner account.

Result

Even with the `require` statement after the function execution, the transactions still fail for non-owners. The `require` statement reverts all changes to the initial state.

Security Vulnerability

Edge Case

There is a subtle security problem related to the order of operations.

Scenario

The modifier checks can prevent unauthorized setting of text and transferring of ownership, but there is an edge case where the contract can be exploited. This is a good example of how hacks can occur.

Layout Order

```
State variables -> Modifiers -> Functions
```

How Hacks Occur

Hackers test thousands of things, and all of them are working. Then, just one edge situation that you didn't think of is not working, and then you can get hacked.

Security Risk: Hijacking Smart Contracts

A critical security risk can arise if the **owner** of a smart contract is set incorrectly. This vulnerability allows malicious actors to take control of the contract. Specifically, overwriting the owner in the last line of code can lead to the contract being hacked.

Even a small change in the code can lead to exploits that could result in significant financial loss.

Testing and Security Practices

Comprehensive testing is essential to identify and mitigate security issues in smart contracts. The lecture will cover testing techniques and tools to ensure code safety and security over the next couple of weeks.

Homework & Environment Setup for Lesson 5

Environment Setup

For lesson five, students will transition from using Remix to coding on their local machines. It is crucial to prepare the development environment beforehand. Here's what you need to have installed:

- **Node.js:** Version 20
- **npm**
- **yarn** (optional)
- **pnpm** (optional)
- **Git CLI**
- **VS Code** (or any similar code editor like Cursus)

It is important to be familiar with using command-line tools and running Node.js and npm commands.

RPC Node Provider Accounts

Create free accounts on the following providers:

- Thirdweb
- Alchemy
- Infura

These accounts will be needed for making RPC calls and interacting with blockchains.

Hardhat

The boot camp will use Hardhat for development.

Weekend Project Details

Project Overview

The weekend project involves interacting with the `hello.sol` smart contract using Remix connected to a test network (e.g., Sepolia).

Steps

1. Interact with the Contract: Work within your assigned group to interact with the `hello.sol` contract. This includes changing the message strings and transferring ownership.
2. Error Observation: Identify and document any errors that occur when interacting with the contract as the owner and as a non-owner.
3. Report Creation: Write a report detailing the execution of each function and transaction hashes. Include observations on scenarios where transactions revert due to permission issues (e.g., calling `transferOwnership` without being the owner).
    - Format: The format of the report is flexible (Google Docs, Notion page, spreadsheet, text file, etc.).
    - Content: Document the execution of each function and transaction hashes to demonstrate activity on the blockchain. Note any instances where transactions revert due to permission issues.

Collaboration

Collaborate with your group members by taking turns deploying the contract and calling functions. This round-robin approach allows everyone to participate and understand the contract's behavior.

Submission

One person from each group should fill out the submission form (link will be provided on the Reach Me channel on Discord) and include the report.

Test Network

Use any test network (Sepolia, Polygon, Coinbase, etc.) or even the mainnet if you prefer.

Group Information

- **Group Size**: Aim for groups of at least three students, with four to seven being ideal.
- **Group Location**: Find your assigned group on Discord. If you are not in a group, contact a program manager ASAP.
- **Contract Verification**: Verifying the contract is optional but recommended.

Extra Reference Material

Prepare for the next week by reviewing TypeScript and JavaScript resources, as the upcoming sessions will heavily utilize these languages.

Summary of tasks

Task	Details
Setting up development environment	Install Node.js v20, npm, yarn (optional), Git CLI, and VS Code (or similar).
Create RPC Node Provider Accounts	Create free accounts on Thirdweb, Alchemy, and Infura.
Weekend project using `hello.sol`	Deploy the `hello.sol` contract in Remix, Interact with the contract to change the message and transfer ownership in your group, and Document errors and transaction hashes.
Submit group report	One person from each group fills the submission form and includes the report.
Review TypeScript and JavaScript resources	Since the upcoming sessions will heavily utilize these languages, it's important to get familiar with them.
window.MathJax = {	

```
  tex: {
    inlineMath: [['$', '$'], ['\\(', '\\)']],
    displayMath: [['$$', '$$'], ['\\[', '\\]']]
  }
```

};