**EVM signatures selectors**

**Advanced EVM - Encoding Signatures & Selectors**

Welcome back! Let's bring it all home by learning how we can populate the data field of our transactions to call any function we want.

Step 1 will be creating a new file in Remix named CallAnything.sol.

Now, in order to execute a function using only the data field of a call we need to encode two things:

1. The function name
2. The function parameters to pass

To do this, we're going to need to work with a couple new concepts.

1. **function selector** - the first 4 bytes of a function signature
   * **Example:** 0xa9059cbb - this is the function selector of a transfer function.
2. **function signature** - a string which defines a function name and its parameters
   * **Example:** "transfer(address,uint256)"

When we send a call to an address, the EVM determines how to respond based on the data we pass with the transaction. We call specific functions by assuring this data includes the function selector we want engaged when the transaction is placed.

One way we can acquire the function selector is to encode the entire function signature, and grab the first 4 bytes of the result. Let's see what this looks like in our contract.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

contract CallAnything {

address public s\_someAddress;

uint256 public s\_amount;

function transfer(address someAddress, uint256 amount){

s\_someAddress = someAddress;

s\_amount = amount;

}

}

The above function will have the exact function signature and function selector we saw in our examples.

**Function Selector:** 0xa9059cbb

**Function Signature:** "transfer(address,uint256)"

This is great when we already know a function selector, but..

***How do we acquire the function selector programmatically?***

The answer is - we can write a function! There are actually a few different ways we can approach this, let's go through them.

function getSelectorOne() public pure returns(bytes4 selector){

selector = bytes4(keccak256(bytes("transfer(address,uint256)")));

}

Adding this to our Remix contract, we can compile and deploy. Calling this function results in...

This is exactly what we'd expect it to be! Great! Now what else do we need? The parameters we're passing our function call are going to need to be encoded with this signature.

Much like abi.encode and abi.encodePacked, the EVM offers us a way to encode our parameters with a given selector through abi.encodeWithSelector

We can write another function to compile this data for our function call for us.

function getDataToCallTransfer(address someAddress, uint256 amount) public pure returns(bytes memory){

return abi.encodeWithSelector(getSelectorOne(), someAddress, amount);

}

If we compile CallAnything.sol and redeploy in Remix, we can call this function now to get all the data required to call the transfer function. Passing getDataToCallTransfer the contracts own address and an amount of 50 outputs:

0:bytes: 0xa9059cbb0000000000000000000000007b96af9bd211cbf6ba5b0dd53aa61dc5806b6ace0000000000000000000000000000000000000000000000000000000000000032

This is the data we would need to pass a low-level call in order to call the transfer function with our given parameters. We can now write a function that uses this data to make the function call.

function callTransferWithBinary(address someAddress, uint256 amount) public returns(bytes4, bool){

(bool success, bytes memory returnData) = address(this).call(abi.encodeWithSelector(getSelectorOne(), someAddress, amount));

}

❗ **PROTIP** We could also use address(this).call(getDataToCallTransfer(someAddress, amount));

In the above we're sending our function call to the contract's own address, but this could be any address technically. This call is going to return two things which we're assigning to success and returnData.

**success:** A boolean value representing if the transaction was successfully completed.

**returnData:** any return data provided as a result of the function call.

Typically we'd see something requiring success to be true, but for our example we'll just have our function return these values.

function callTransferWithBinary(address someAddress, uint256 amount) public returns(bytes4, bool){

(bool success, bytes memory returnData) = address(this).call(abi.encodeWithSelector(getSelectorOne(), someAddress, amount));

return(bytes4(returnData), success);

}

What makes this so powerful is the ability to send transaction data this way, agnostic of the contract you send it to. All you need is to change address(this) to the address you want to send the data to.

❗ **NOTE** This doesn't mean all addresses receiving the data will know what to do with it!

Let's run this function in Remix to see it in action. Compile and redeploy CallAnything.sol.

As expected, after deployment our storage variables initialize as 0

Now, if we pass the contract address and 50 as an amount to our callTransferWithBinary function, Remix's terminal should provide us an output on what happened.

Here we can see that our transaction was successful, represented by the bool true. The bytes4 value of our returnData is empty, because our transfer function doesn't actually return anything!

With this transaction complete, we should be able to repoll the storage variables in our contract. We would expect them to be updated with the values we passed callTransferWithBinary...

...and they are! Amazing! Another option Solidity affords us is the ability to encode with a signature. This effectively saves us a step since we don't have to determine the function selector first.

function callTransferWithBinarySignature(address someAddress, uint256 amount) public returns(bytes4, bool){

(bool success, bytes memory returnData) = address(this).call(abi.encodeWithSignature("transfer(address,uint256)", someAddress, amount));

return(bytes4(returnData), success);

}

This should behave the exact same, try it out yourself!

**Alternative Selector Acquisition**

I mentioned there were a few different ways to acquire a function selector and there are a multitude of possible reasons why you may leverage one method over another

We wont walk through all the different methods here, but I've provided some of them below and these are also available in the course's [**GitHub repo**](https://github.com/Cyfrin/foundry-nft-f23/blob/main/src/sublesson/CallAnything.sol).

CallAnything.sol

// SPDX-License-Identifier: MIT

// So why do we care about all this encoding stuff?

// In order to call a function using only the data field of call, we need to encode:

// The function name

// The parameters we want to add

// Down to the binary level

// Now each contract assigns each function it has a function ID. This is known as the "function selector".

// The "function selector" is the first 4 bytes of the function signature.

// The "function signature" is a string that defines the function name & parameters.

// Let's look at this

pragma solidity 0.8.20;

contract CallAnything {

address public s\_someAddress;

uint256 public s\_amount;

function transfer(address someAddress, uint256 amount) public {

// Some code

s\_someAddress = someAddress;

s\_amount = amount;

}

// We can get a function selector as easy as this.

// "transfer(address,uint256)" is our function signature

// and our resulting function selector of "transfer(address,uint256)" is output from this function

// one thing to note here is that there shouldn't be any spaces in "transfer(address,uint256)"

function getSelectorOne() public pure returns (bytes4 selector) {

selector = bytes4(keccak256(bytes("transfer(address,uint256)")));

}

function getDataToCallTransfer(address someAddress, uint256 amount) public pure returns (bytes memory) {

return abi.encodeWithSelector(getSelectorOne(), someAddress, amount);

}

// So... How can we use the selector to call our transfer function now then?

function callTransferFunctionDirectly(address someAddress, uint256 amount) public returns (bytes4, bool) {

(bool success, bytes memory returnData) = address(this).call(

// getDataToCallTransfer(someAddress, amount);

abi.encodeWithSelector(getSelectorOne(), someAddress, amount)

);

return (bytes4(returnData), success);

}

// Using encodeWithSignature

function callTransferFunctionDirectlyTwo(address someAddress, uint256 amount) public returns (bytes4, bool) {

(bool success, bytes memory returnData) =

address(this).call(abi.encodeWithSignature("transfer(address,uint256)", someAddress, amount));

return (bytes4(returnData), success);

}

// We can also get a function selector from data sent into the call

function getSelectorTwo() public view returns (bytes4 selector) {

bytes memory functionCallData = abi.encodeWithSignature("transfer(address,uint256)", address(this), 123);

selector =

bytes4(bytes.concat(functionCallData[0], functionCallData[1], functionCallData[2], functionCallData[3]));

}

// Another way to get data (hard coded)

function getCallData() public view returns (bytes memory) {

return abi.encodeWithSignature("transfer(address,uint256)", address(this), 123);

}

// Pass this:

// 0xa9059cbb000000000000000000000000d7acd2a9fd159e69bb102a1ca21c9a3e3a5f771b000000000000000000000000000000000000000000000000000000000000007b

// This is output of `getCallData()`

// This is another low level way to get function selector using assembly

// You can actually write code that resembles the opcodes using the assembly keyword!

// This in-line assembly is called "Yul"

// It's a best practice to use it as little as possible - only when you need to do something very VERY specific

function getSelectorThree(bytes calldata functionCallData) public pure returns (bytes4 selector) {

// offset is a special attribute of calldata

assembly {

selector := calldataload(functionCallData.offset)

}

}

// Another way to get your selector with the "this" keyword

function getSelectorFour() public pure returns (bytes4 selector) {

return this.transfer.selector;

}

// Just a function that gets the signature

function getSignatureOne() public pure returns (string memory) {

return "transfer(address,uint256)";

}

}

One last thing I want to point out is that we're not limited to this kind of interaction. Through this low-level calling method, two contracts are able to interact without possessing all the information associated with each other. Consider this second contract CallFunctionWithoutContract.

CallFunctionWithoutContract

contract CallFunctionWithoutContract {

address public s\_selectorsAndSignaturesAddress;

constructor(address selectorsAndSignaturesAddress) {

s\_selectorsAndSignaturesAddress = selectorsAndSignaturesAddress;

}

// pass in 0xa9059cbb000000000000000000000000d7acd2a9fd159e69bb102a1ca21c9a3e3a5f771b000000000000000000000000000000000000000000000000000000000000007b

// you could use this to change state

function callFunctionDirectly(bytes calldata callData) public returns (bytes4, bool) {

(bool success, bytes memory returnData) =

s\_selectorsAndSignaturesAddress.call(abi.encodeWithSignature("getSelectorThree(bytes)", callData));

return (bytes4(returnData), success);

}

// with a staticcall, we can have this be a view function!

function staticCallFunctionDirectly() public view returns (bytes4, bool) {

(bool success, bytes memory returnData) =

s\_selectorsAndSignaturesAddress.staticcall(abi.encodeWithSignature("getSelectorOne()"));

return (bytes4(returnData), success);

}

function callTransferFunctionDirectlyThree(address someAddress, uint256 amount) public returns (bytes4, bool) {

(bool success, bytes memory returnData) = s\_selectorsAndSignaturesAddress.call(

abi.encodeWithSignature("transfer(address,uint256)", someAddress, amount)

);

return (bytes4(returnData), success);

}

}

By passing this contract the address of our CallAnything.sol deployment. We're able to use the functions it possesses to interact with CallAnything.sol

Before we interact with anything, recall what the values of our storage variables on CallAnything.sol are currently.

Now we can call callTransferFunctionDirectlyThree on our CallFunctionWithoutContract.sol by passing a new address and amount. This should result in an updating of the storage variables on CallAnything.sol via this low-level call.

**Wrap Up**

Hopefully by now you can see the power available through this methodology of low-level calls. Now, despite hyping it up for several lessons, low-level calls are risky, and it's worth noting that they should be avoided when possible. Use an interface or something similar if you can, because low-level calls can leave you open to a number of potential issues and vulnerabilities.

With that said, you've just learnt some really advanced stuff. If it's a little confusing, don't feel bad, you can always come back later when you've gained a little more experience and context of the EVM.

If you're excited to learn more about how Solidity works under-the-hood, I recommend reading through the [**Deconstructing Solidity**](https://blog.openzeppelin.com/deconstructing-a-solidity-contract-part-i-introduction-832efd2d7737) series by OpenZeppelin. It does a great job breaking things down in a very digestible and granular way.

With that said, we're almost done, We've a couple things to tidy up in the section. Let's finish strong.