**Solidity ABI Encoding and Decoding Functions**

**Table of Contents**

* ABI encoder
* ABI decoder
* ABI Packed Encoding

**ABI Encoding and Decoding**

A smart contract is primarily made up of state variables and functions. Some functions are private and can only be accessed from within the contract, although many functions are public and may be used from outside the contract. That is, programs (and humans) can submit data to the contract and retrieve data from the contract.

To submit data to the contract, we need to send it in a way that the contract can read it. That is, they need to be encoded. The rule on how to execute such encoding is defined by the implementation of the Ethereum Virtual Machine (EVM) (EVM)

# ABI Encoder

Solidity contains a global variable named abi that has an encode method, which means that we can use it to encode the parameters of any function that we write. Let's start with a straightforward illustration. Let's pretend we have the following function available.

| function myFunction(address \_myAddress, uint \_myNumber)... |
| --- |

We are just concerned with encoding the parameters of the function, namely, an address and an integer, and nothing else. We may use the remix function to create a function that does this function.

| pragma solidity ^0.8.0;  contract Encode { function encode(address \_address, uint \_int) public returns(bytes memory) {  return (abi.encode(\_address, \_int)); } } |
| --- |

After deploying this contract and using the function encode(...) with the following values for the address and unsigned integer:

(0x5B38Da6a701c568545dCfcB03FcB875f56beddC4, 127)

we obtain the following result:

| 0x0000000000000000000000005b38da6a701c568545dcfcb03fcb875f56beddc4000000000000000000000000000000000000000000000000000000000000007f |
| --- |

A quick examination of the result reveals that it is composed of 64 bytes. This is due to the fact that the encoding is done in multiples of 32 bytes:

The address (20 bytes) is contained inside the first 32 bytes, while the integer 7f is included within the last 32 bytes. Encoding is usually done in hexadecimal, thus the number 7f is represented by the number 127.

ABI Decoder

Let's have a look at how to decode the parameters of a function using solidity. It should be noted that it was not essential to identify the function with which we were dealing because the signature of the function would appear before the encoded parameters. Using dynamic variables as parameters will add a small layer of complexity to the issue solving process.

Let's decode a tuple of values consisting of string, uint, and string using the following contract.

| contract Encode { function encode(string memory \_string1, uint \_uint, string memory \_string2) public pure returns (bytes memory) {  return (abi.encode(\_string1, \_uint, \_string2));  } function decode(bytes memory data) public pure returns (string memory \_str1, uint \_number, string memory \_str2) {  (\_str1, \_number, \_str2) = abi.decode(data, (string, uint, string));   } } |
| --- |

Deploying, then calling the function encode(…) with the following parameters (Ineuron, 3, Oneneuron), we have the following return:

0x0000000000000000000000000000000000000000000000000000000000000060000000000000000000000000000000000000000000000000000000000000000a00000000000000000000000000000000000000000000000000000000000000a00000000000000000000000000000000000000000000000000000000000000007496e6575726f6e0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000094f6e656e6575726f6e0000000000000000000000000000000000000000000000

In this case, you may anticipate that the result would be 96 bytes (three times 32), given that we have three variables. However, two of the three variables are dynamic, and encoding dynamic variables is not as straightforward as it might seem. Let us break down the above value into 32 byte chunks of data for your convenience.

0000000000000000000000000000000000000000000000000000000000000060

000000000000000000000000000000000000000000000000000000000000000a

00000000000000000000000000000000000000000000000000000000000000a0

0000000000000000000000000000000000000000000000000000000000000007

496e6575726f6e000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000009

4f6e656e6575726f6e00000000000000000000000000000000000000000000000

Consider a brief description of how string encoding is accomplished in the context of string data. The first line refers to the first variable, the second to the second variable, and the third to the third variable. Because the second variable is of the type value, we can directly access its value, which is 10(the hexadecimal equivalent of 10 is a), in the second line because it is of type value.

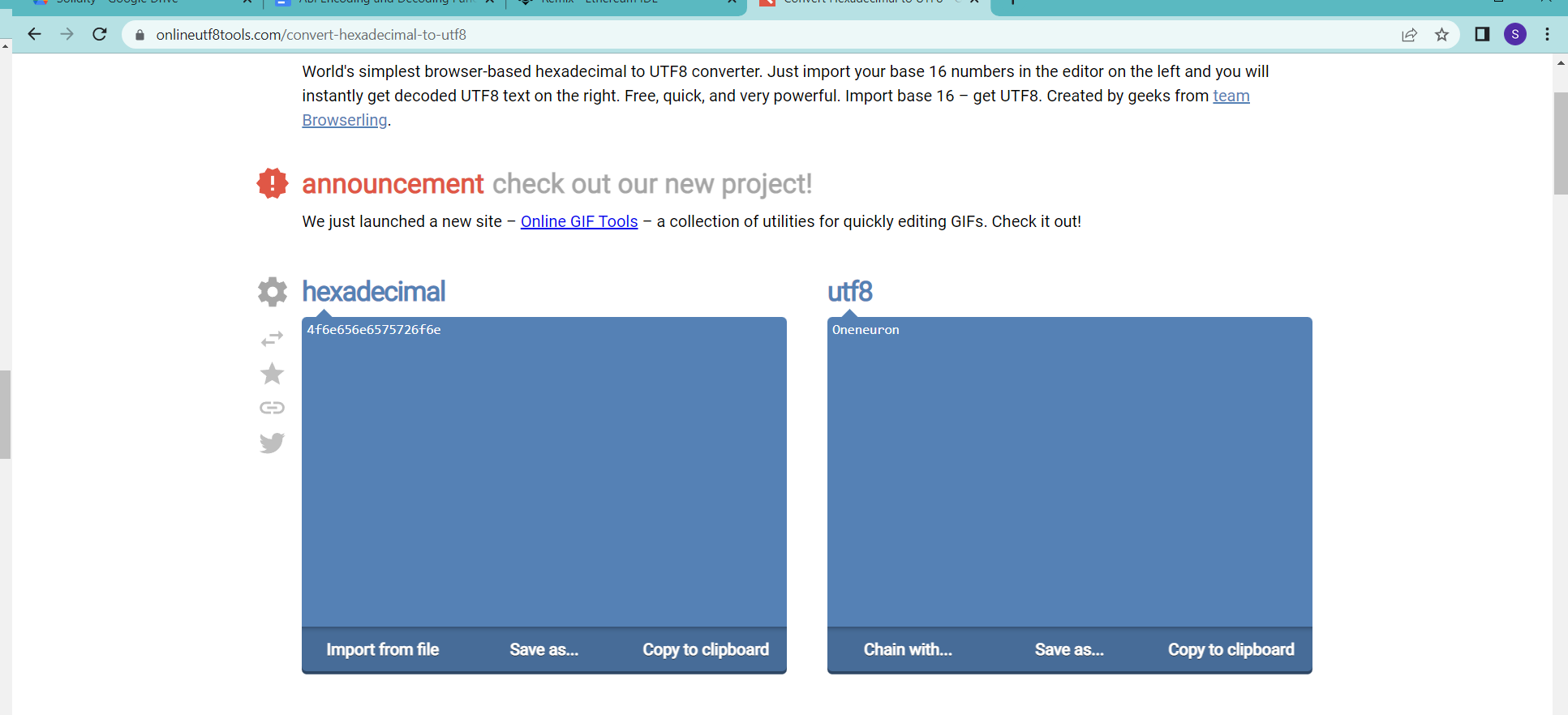
As an example, consider a variable-type string. The data contained in the first line is information about the first string, and the data contained in the second line is information about the second string. Using the data in this example, the value is 60 in hexadecimal, which corresponds to 96 in decimal. But what exactly does this mean? In other words, the information about the first string is discovered 96 bytes after the beginning of the data is received.

There is a number 7 after the first 92 bytes of the chunk of 32 bytes. This is the number of bytes that the text occupies, starting on the following line and encoded in utf-8: 496e6575726f6e, starting on the next line. When we convert the word 'Ineuron' from hex to UTF-8, we get the word 'Ineuron' back .

Following the same pattern, the third string can be retrieved after a0 bytes, which is 160 bytes after the beginning of the data was received. It also states that it has 9 bytes, and that its value is 4f6e656e6575726f6e which is the UTF-8 encoding for the word 'Oneneuron' .

You can try the following link and manually verify the hexadecimal values for Ineuron and Oneneuron

<https://onlineutf8tools.com/convert-hexadecimal-to-utf8>



The screenshot above shows the hexadecimal value - 4f6e656e6575726f6e and corresponding string value- Oneneuron.

**ABI encodePacked**

Solidity includes a built-in function abi.encodePacked(...) that allows for the encoding of data in a non-standard manner. This enables data to be encoded as raw bytes without having to adhere to the ABI's rules for encoding data.

When doing packed encoding in Solidity, the following ABI restrictions are not enforced anymore:

Dynamic types (for example, strings, arrays, and so on) are represented exactly as they are, without any offset or length information.

Zero padding is not applied to static types that are shorter than 32 bytes (for example, uint8, bytes4, and so on).

Let's take an example of encoding two strings text1 and text2 (“AAAA”,”BBBB”) using abi.encode and abi.ecnodePacked and compare the results for further analysis:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Hashfunction{

function hash (string memory text,uint num,address addr)external pure returns(bytes32){

return keccak256(abi.encodePacked(text,num,addr));

}

function encode(string memory text1, string memory text2) external pure returns(bytes memory){

return abi.encode(text1,text2);

}

function encodepacked(string memory text1, string memory text2) external pure returns(bytes memory){

return abi.encodePacked(text1,text2);

}

function collision(string memory text1, string memory text2) external pure returns(bytes32)

{

return keccak256(abi.encodePacked(text1,text2));

}

}

Upon executing the functions, you can clearly notice that abi.encodePacked is susceptible to hash collisions(different inputs giving the same result or output). This can be solved by using something like an integer along with the aforementioned strings which can act like a salt phrase and get rid of hash collisions.