Feb 4, 2023 5 min read

Solidity Gasleft

Updated: Mar 29, 2023

Introduction

The purpose of this article is to describe the behavior of the solidity gasleft() function and its uses.

It is a built-in function that is used to check the remaining gas during a contract call. It is one of the special variables and functions that always exist in the global namespace and thus does not need to be imported. gasleft() was previously known as msg.gas before Solidity version 0.4.21(msg.gas)

Authorship

This article was co-written by Jesse Raymond (LinkedIn, Twitter) as part of the RareSkills Technical Writing Program.

Why gasleft() matters

The amount of gas used by smart contracts depends on the complexity of the code being run and the amount of data that is being processed during the contract call.

If the provided gas isn't enough, the transaction reverts with an "out of gas" error. Proper use of the gasleft() function can prevent situations where contract transactions run out of gas. Let's look at an example in the next section.

Example of preventing out-of-gas error

Not running out of gas while distributing Ether

Sending Ether to multiple addresses in smart contracts via loops can be very expensive, especially when dealing with a large array of addresses.

If the amount of gas used to execute the transaction is not sufficient, the function will fail with an "out of gas" error, as stated earlier.

However, the gasleft() function can be used to ensure that the remaining gas is sufficient to conduct the next transfer, and exit early otherwise.

The following code demonstrates this

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;
contract GasConsumer{
```

```
    uint constant MINIMUM_AMOUNT = 10_000;

    // this is for illustration purposes only, not production
    function distributeEther(address[] calldata receivers) external {
        for (uint i = 0; i < receivers.length; i++) {

            payable(receivers[i]).transfer(1 ether);

            if (gasleft() < MINIMUM_AMOUNT) {
                return;
            }
        }
    }

    receive() external payable {}
}
```

The function "distributeEther" in the contract above takes an array of addresses, iterates through the array using a for loop, and sends 1 ether to each address with the "transfer" function.

The "if statement" checks if the remaining gas after each Ether transfer in the loop is sufficient for the next transfer by checking whether the "gas left" after a transfer is less than 10,000 (9,000 for transferring Eth and an extra 1,000 for other minor opcodes). If it is less, the transaction ends without reverting the previous transfers.

(Note that it is not a good idea to push out Ether unless the address are trusted. A malicious receiver could submit an address of a smart contract that reverts when receiving ether).

Benchmarking code

Using the gasleft() function to measure execution cost

Another example is using the gasleft() function to measure the total amount of gas used by a section of code.

Here's an example in remix:

benchmark solidity code with gasleft

In this case, the gasleft() function is used to find out how much gas is being used when a number is appended to the “numArr” array using the “updateArray” function. This is not the total amount of gas used by the function, but the amount of gas used before and after a number is appended to the array at the code numArr.push(_num) at line 30.

Explaining the highlighted numbers

We set “gasUsed” to be a public variable, so we can easily see the contents after the function is executed. A test for this is done by deploying the “GasCalc” contract and calling the “updateArray” function.

The function returns a tuple, with the first entry resulting in **80,348**, which is the “initialGas”.
The second entry in the tuple is “finalGas” and it was **35,923**, **this includes the gas cost for the gasleft() function itself**.

By subtracting the final gas from the initial gas, we determine the execution cost of line 30 is **44,425**.

The opcode behind gasleft requires “2 gas” to execute

Solidity is a high-level language that is compiled into bytecodes which are executed on the Ethereum Virtual Machine(EVM).

The opcode for the gasleft() function is GAS (bytecode 0x5A), which costs "2 gas" according to the ethereum documentation.

Real-World Applications

OpenZeppelin proxy – used to forward all gas to the implementation contract

Using gasleft in yul

The gasleft() function can also be accessed in Solidity smart contracts via yul (inline assembly), as gas().

The OpenZeppelin proxy contract is an excellent example of how this is done. It is used in the "delegatecall" function which the proxy uses to call the implementation contract. gas() is a convenient way to specify using the maximum available gas for this operation.

forward all gas in delegate call with gasleft solidity

Link: OpenZeppelin Proxy Contract

OpenZeppelin Minimal Forwarder – used to validate the relayer sent enough gas to execute the transaction

A “Relayer” is an off-chain entity that pays for the gas of another user’s transactions and the transaction is sent to a “Forwarder” contract which executes the transaction.
When a user sends a request to the relayer, the user specifies the amount of gas to include in the transaction, and digitally signs their request.

However, the relayer might not respect the gas limit requested by the user, and send a lower amount. This attack is documented in the SWC Registry as SWC-126.

This causes a gas-griefing attack. If the relayer’s call to the forwarding contract succeeds, but the sub call the user wants fails, then the relayer can “blame” the user for sending a transaction that reverts when the real reason was the subcall ran out of gas due to the relayer not sending enough.

A subcall can fail due to a revert or an out-of-gas error, but the failure reason is generally not given, just the boolean success variable returns false. Because of this, we don’t know if the subcall failed due to insufficient gas or bad instructions from the original sender.
We can use “gasleft” to determine which case it is.

When the execution to “call” happens, only 63/64 of the gas is forwarded. This 63/64 limit was introduced in EIP 150 and you can read more about it here.

After the subcall completes, then the amount of gas left should be at least 1/64 of the original limit specified by the user.

If there is less than 1/64 of the requested gas after the subcall, then we know the relayer didn’t send all of the gas they were supposed to.

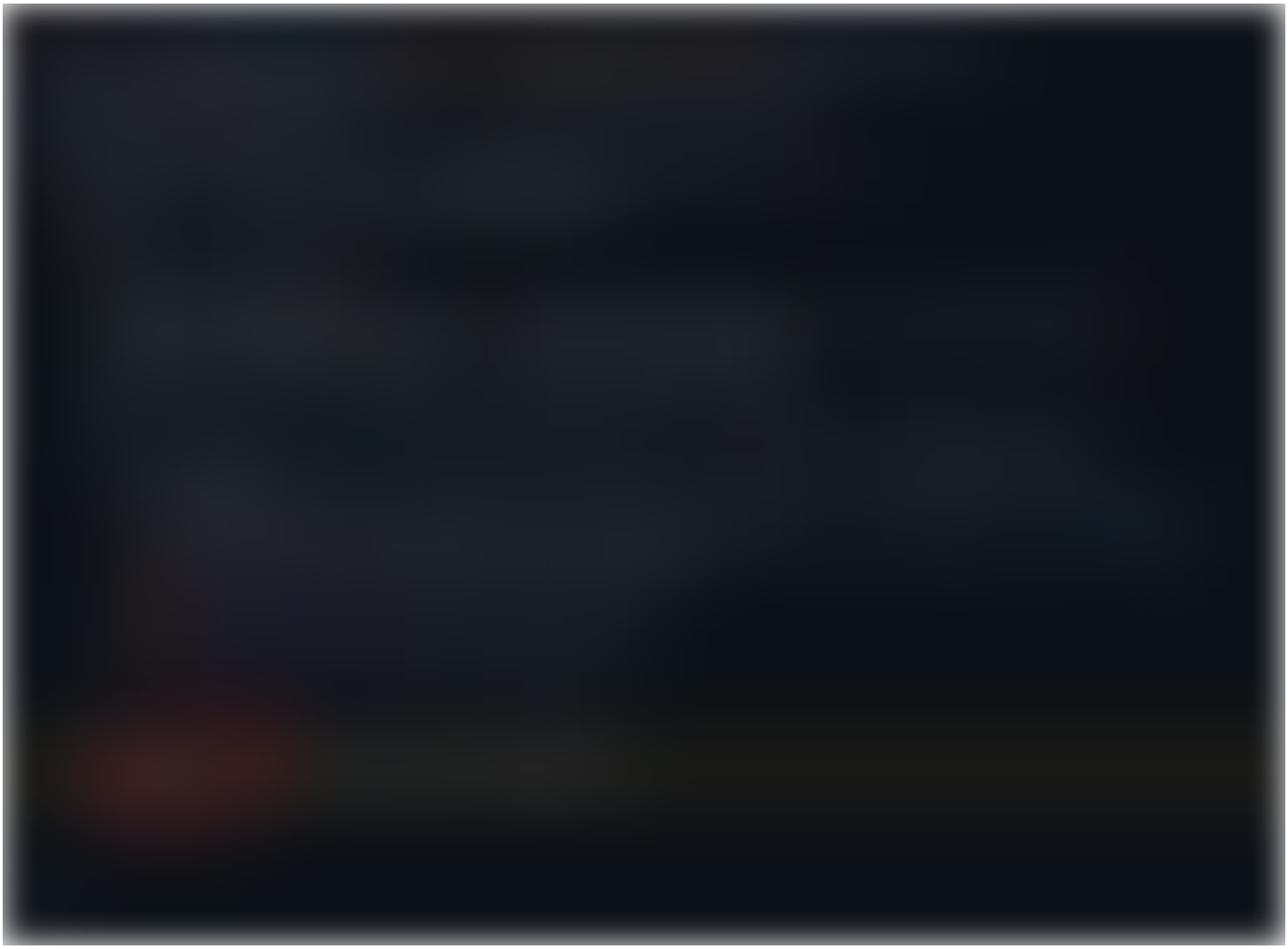
The forwarding contract here checks if at least 1/63 of the original limit is left as a margin of safety.

The invalid code is used to cause the relayer’s transaction to fail to make it clear the transaction failed due to the relayer, not the subcall. You can read more about the gas griefing attack here.

Link: OpenZeppelin Minimal Forwarder Contract

Chainlink EthBalance Monitor Contract – used to prevent the out-of-gas error from blocking Ether distribution

This is a real-life application of the first example in this article, where we distributed ether in a loopp. There is more business logic in this code compared to earlier, but if we highlight the "gasleft()" check that causes an early exit from the loop, we see it is fundamentally the same design.



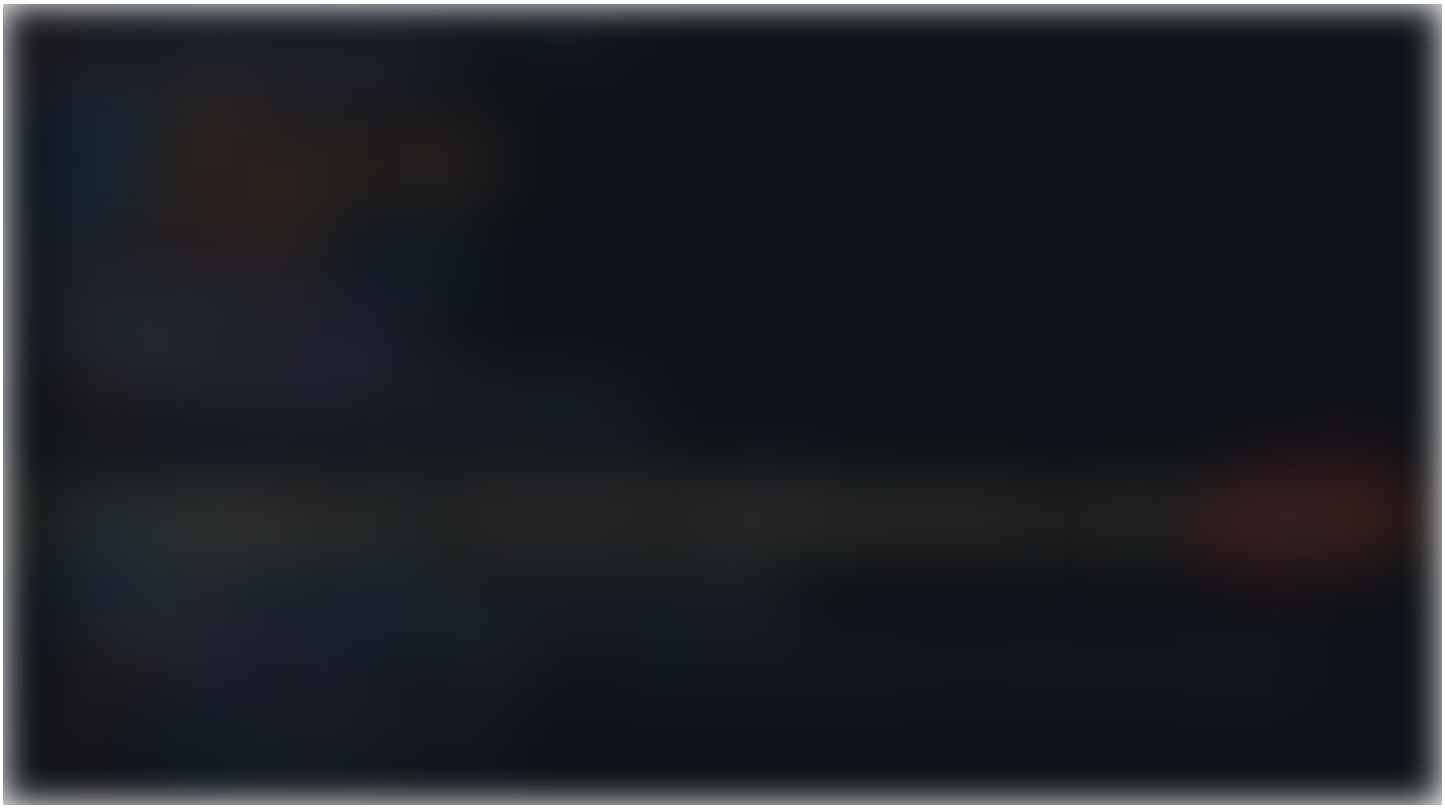
Chainlink VRFCoordinatorV2 Contract – used to get the amount of gas used for Chainlink VRFCoordinatorV2 fulfillments

The Chainlink "VRFCoordinatorV2" smart contract is a "Verifiable Random Function" coordinator, used for generating cryptographically secure random numbers on the blockchain.

The smart contract is the oracle for the smart contract to request and receive randomness. (see here: VRFCoordinatorV2).

The gasleft() function is used in the contract's "calculatePaymentAmount" function to charge the user a larger fee if the nodes need to pay more gas to fulfill the randomness.

The lower gasleft() is in this circumstance, the higher the fee will be because (startGas - gasleft()) increases as more gas is consumed.



Link: Chainlink VRFCoordinatorV2 Contract

Conclusion

In this article, we have discussed various use cases for gasleft(). These include preventing out-of-gas errors, benchmarking solidity code execution cost, forwarding all gas to implementation contracts, and preventing relayer DOS.

RareSkills Blockchain Bootcamp

Please see our advanced blockchain bootcamp offerings to learn more about the expert-level developer training we offer.