Minimal Transaction Bundler

shadowysupercoder.eth

December 2021

1 Introduction

Some actions on the blockchain require multiple transactions, e.g. calling an *approve* on a token contract before calling *deposit* on a staking contract. This document describes the Minimal Transaction Bundler (hereafter referred to simply as Bundler), a gas optimized smart contract wallet on EVM compatible blockchains that allows an EOA to submit bundles of transactions, saving on the overhead costs of submitting each transaction separately.

2 Properties

1. Bundles are all-or-nothing. If a single call fails, the whole transaction is reverted.
2. If a transaction returns data, then the return data is emitted in a raw log event, but Bundler does not otherwise handle return data. This means that all non-reverting calls are taken to be successful.
3. Each call in a bundle makes use of a simple compression protocol that limits ETH transfers to ≈ 1.20 million ETH and limits calldata to ≈ 65 kilobytes.
4. Bundler assumes calls are properly encoded. Incorrectly encoded calls could have unknown consequences, including up to total loss of funds.

3 Encoding

3.1 Individual Calls

Three main parts compose any transaction call accepted by Bundler.

1. target
   - The 20 bytes address of an Ethereum account.
2. sendValue
   - The amount of ETH to send with the call.
3. calldata
   - The data to send with the call.

The first 32 bytes of an encoded call contains the target address as 20 bytes (or uint160), the sendValue as 10 bytes (or uint80), and the size of the calldata as 2 bytes (or uint16).

$$data[0:32] := (target) + (sendValue) + (size)$$

If there is no calldata, then the call is fully encoded in those 32 bytes (and $size := 0$). Otherwise, append the calldata to the first 32 bytes.

$$data[32:32+size] := calldata$$

The final result is that each call is encoded as

$$data := (target) + (sendValue) + (size) + [calldata]$$

where the square brackets indicate that calldata is optional.

3.2 Bundled Calls

Bundles are the concatenation of individual calls. There's no need to encode a total length for the bundle itself, because Bundler uses the EVM opcode CALLDATASIZE to find the end of the bundle.

3.3 Example

Suppose Bundler owns 1000 LINK tokens, and the controller of Bundler wants to both approve and deposit 1000 LINK into a staking contract in one transaction.

Let the LINK token address be its Ethereum mainnet address

$$\texttt{0x514910771af9ca656af840dff83e8264ecf986ca}$$

Let the address of the staking contract be

$$\texttt{0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef}$$

The function selector for the approve function is derived from its function signature: approve(address spender, uint256 value)

$$\texttt{0x095ea7b3}$$

The function selector for the deposit function is derived from its function signature: deposit(uint256 value)

$$\texttt{0xb6b55f25}$$

3.3.1 Encoding the approve call

For the approve(address spender, uint256 value) call we have:

$$\begin{aligned}
\text{target} &:= \texttt{0x514910771af9ca656af840dff83e8264ecf986ca} \\
\text{sendValue} &:= \texttt{0x0000000000000000000} \\
\text{calldata} &:= \texttt{0x095ea7b300000000000000000000000deadbeefdeadbeefdeadbeefdeadbeef} \\
&\quad \texttt{deadbeef0000000000000000000000000000000000000000000003635c9adc5} \\
&\quad \texttt{dea00000} \\
\text{size} &:= \texttt{0x0044}
\end{aligned}$$

The first 32 bytes of the encoded approve call are

$$\begin{aligned}
data[0:32] &:= (target) + (sendValue) + (size) \\
data[0:32] &:= (\texttt{0x514910771af9ca656af840dff83e8264ecf986ca}) \\
&\quad + (\texttt{0x0000000000000000000}) \\
&\quad + (\texttt{0x0044}) \\
data[0:32] &:= \texttt{0x514910771af9ca656af840dff83e8264ecf986ca0000000000000000000000044}
\end{aligned}$$

Then we concatenate the calldata from above to obtain

$$\begin{aligned}
\text{encodedApproveCall} := \texttt{0x514910771af9ca656af840dff83e8264ecf986ca0000000000000000000000044} \\
\texttt{095ea7b300000000000000000000000deadbeefdeadbeefdeadbeefdeadbeef} \\
\texttt{deadbeef0000000000000000000000000000000000000000000003635c9adc5} \\
\texttt{dea00000}
\end{aligned}$$

3.3.2 Encoding the deposit call

For the deposit(uint256 value) call we have:

$$\begin{aligned}
\text{target} &:= \texttt{0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef} \\
\text{sendValue} &:= \texttt{0x0000000000000000000} \\
\text{calldata} &:= \texttt{0xb6b55f250000000000000000000000000000000000000000000003635c9adc5} \\
&\quad \texttt{dea00000} \\
\text{size} &:= \texttt{0x0024}
\end{aligned}$$

The first 32 bytes of the encoded deposit call are

$$
\begin{aligned}
data[0:32] &:= (target) + (sendValue) + (size) \\
data[0:32] &:= (\texttt{0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef}) \\
&\quad + (\texttt{0x0000000000000000000}) \\
&\quad + (\texttt{0x0024}) \\
data[0:32] &:= \texttt{0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef0000000000000000000000024}
\end{aligned}
$$

Then we concatenate the calldata from above to obtain

$$
\begin{aligned}
\text{encodedDepositCall} := \texttt{0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef0000000000000000000000024} \\
\texttt{b6b55f250000000000000000000000000000000000000000000003635c9adc5} \\
\texttt{dea00000}
\end{aligned}
$$

3.3.3 Final Bundle

The data of the final bundle is the concatenation of the two encoded calls

$$
\begin{aligned}
\text{encodedBundle} := \texttt{0x514910771af9ca656af840dff83e8264ecf986ca0000000000000000000000044} \\
\texttt{095ea7b300000000000000000000000deadbeefdeadbeefdeadbeefdeadbeef} \\
\texttt{deadbeef0000000000000000000000000000000000000000003635c9adc5} \\
\texttt{dea00000deadbeefdeadbeefdeadbeefdeadbeef0000000000000000} \\
\texttt{00000024b6b55f250000000000000000000000000000000000000000000036} \\
\texttt{35c9adc5dea00000}
\end{aligned}
$$

4 Contract Code

Bundler is written in Yul.

```
object "Bundler" {
    code {
        sstore(0, caller())
        datacopy(0, dataoffset("main"), datasize("main"))
        return(0, datasize("main"))
    }
    object "main" {
        code {
            /* revert if caller is not owner */
            if iszero(eq(sload(0), caller())) { revert(0, 0) }

            /*
                Bundler loop.
                The variable `p` is a pointer to the current location in the calldata
                array. The variable `end` is the pointer location where the calldata
                array ends. Each iteration of the loop increments `p` until the end of
                the array is found.
            */
            for {
                let p := 0
                let end := calldatasize()
            } lt(p, end) {
                p := add(p, 32)
            } {
                /* Retrieve target address */
                mstore(0, 0)
                calldatacopy(12, p, 20)
                let target := mload(0)
```

```
                /* Retrieve ETH value */
                mstore(0, 0)
                calldatacopy(22, add(p, 20), 10)
                let sendValue := mload(0)

                /* Retrieve calldata size */
                mstore(0, 0)
                calldatacopy(30, add(p, 30), 2)
                let size := mload(0)

                /* If size is nonzero, store the data to memory and increment `p` */
                if iszero(iszero(size)) {
                    calldatacopy(0, add(p, 32), size)
                    p := add(p, size)
                }

                /* Attempt the call. */
                if iszero(call(gas(), target, sendValue, 0, size, 0, 0)) {
                    revert(0, 0)
                }

                /* If there is return data, then log that data. */
                let returnsize := returndatasize()
                if iszero(iszero(returnsize)) {
                    returndatacopy(0, 0, returnsize)
                    log0(0, returnsize)
                }
            }
        }
    }
}
```