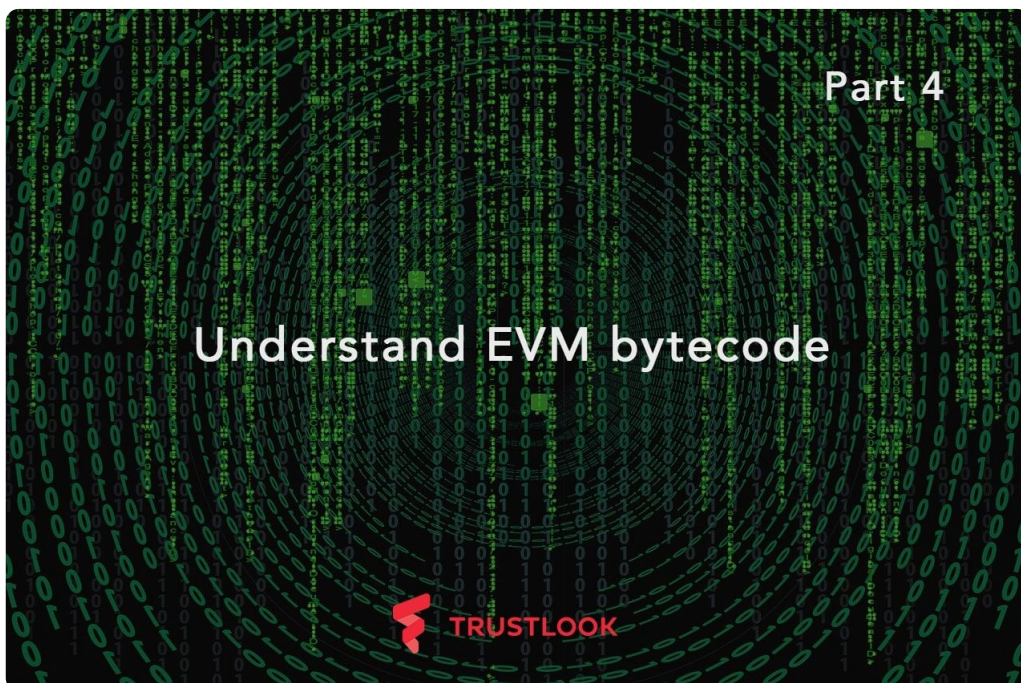




January 9, 2019

Understand EVM bytecode – Part 4



In previous section:

[Understand EVM bytecode – Part 1](https://blog.trustlook.com/understand-evm-bytecode-part-1/)

Understand EVM bytecode – Part 2

Understand EVM bytecode – Part 3

We have talked about how different Solidity data types are implemented in storage. For this section we will dig more about memory and its usage in external calls in this section.

We have learned some basics about memory from previous sections. We know memory is designed for HASH calculation or interactions with external calls or returns. The memory structure has reserved 0x0 and 0x20 for HASH calculation. At address 0x40 it will store a free memory pointer for future use. When some memory need to be allocated, the pointer can be adjusted accordingly so the allocated memory won't be re-visited again. Also, Memory only accessible during contract execution. Once execution is finished, its contents are discarded. Comparing to storage, it is more like the RAM in computer.

Let's look at all the opcodes which depends on memory besides **MLOAD** and **MSTORE**:

```
SHA3
CALLDATACOPY
CODECOPY
EXTCODECOPY
RETURNDATACOPY
LOG1, LOG2, LOG3, LOG4
CREATE
CALL
CALLCODE
RETURN
DELEGATECALL
STATICCALL
REVERT
```

We can see besides the SHA3 is used for calculate HASH value, most of the rest opcodes are related to interactions with EVM. It includes

loading data from EVM data payload or code , or arranging data for external calls and returns.

First, let's look at opcode **CALLDATACOPY**. From Solidity document, "calldatacopy(t, f, s)" is defined as "copy s bytes from calldata at position f to mem at position t". If you got experience to analyze bytecode level contracts, you may observe that another similar opcode **CALLDATALOAD** is more popular than this one. But the difference of these 2 opcodes is that **CALLDATALOAD** only load 32-bytes data into the stack instead of memory. If the contract public function only uses integers in their arguments, then **CALLDATALOAD** is good enough for the calls. The format of the data payload will be as follows:

```
0x00: <function signature hash>
0x04: <first integer argument if any>
0x24: <second integer argument if any>
...
```

However, there are exceptions. Functions can support more data types in Solidity other than pure integers. For example of a function with struct or fixed size arrays:

```
contract Data7 {
    address a;
    address b;

    function test(address [2] addresses) public returns (bool) {
        a = addresses[0];
        b = addresses[1];
        return true;
    }
}
```

When you look at the bytecode generated from above code, you can see :

```
temp0 = mload(0x40);
mstore(0x40, (0x40 + temp0));
calldatacopy(temp0, 0x4, 0x40);
```

Apparently, this time **CALLDATACOPY** is used to copy the whole argument into the memory for future use. It is not just fixed size of arrays. For any parameter which has fixed size like struct. **CALLDATACOPY** will be the one for the work.

However, things can be even more complicated when there is dynamic arrays. For example:

```
contract Data8 {
    address a;
    address b;

    function test(address [] addresses) public returns (bool) {
        a = addresses[0];
        b = addresses[1];
        return true;
    }
}
```

We can see there is only one function *test()* uses an address array as the argument. So how the data will be arranged inside the data payload? Let's still go into the bytecode for the truth. Here is the code snippet before calling the *test()* function:

```
temp0 = mload(0x40);
temp1 = msg.data(0x4)
mstore(0x40, (0x20 + (temp0 + (msg.data((0x4 + temp1)) * 0x20)));
mstore(temp0, msg.data((0x4 + temp1)));
calldatacopy((temp0 + 0x20), (0x24 + temp1), (msg.data((0x4 + temp1))));
var1 = test(temp0);
```

It might not be that obvious to get the logic of the data payload structure. But don't worry. Let's go through it together. The first line *"temp0 = mload(0x40);*

" is a very popular one. It gets the free memory pointer from memory address 0x40 into variable *temp0*. Then, *temp1* will be assigned with the value get from data payload at offset 0x4, which regularly hold the first parameter when the type is integer. However, apparently it is not finished yet in this case. This value in *temp1* will be used as an offset to locate the data starting from 0x4. This value can be shown as *"msg.data((0x4 + temp1))"*. From above code snippet, this value is the length of the array. The size of each item in the array is 0x20. So *"mstore(0x40,(0x20 + (temp0 + (msg.data((0x4 + temp1)) * 0x20))))"* will adjust the free memory pointer to save a piece of memory for this argument. Then, the array length will be copied into the old free memory pointer and items of array will be copied too by **CALLDATACOPY**. Finally, the memory pointer will be transferred to *test()* function for operation.

After we have some basic idea how EVM data payload was arranged, we can see how **CALL** related opcodes work and how memory is involved in. Solidity document states *"call(g, a, v, in, insize, out, outsize) – call contract at address a with input mem[in..(in+insize)) providing g gas and v wei and output area mem[out..(out+outsize)) returning 0 on error (eg. out of gas) and 1 on success"*. So when you use this opcode to call other smart contract you need to supply all of the information. Let's look at a real example of external contract reference:

```
pragma solidity ^0.4.18;
contract Deployed {
    function a() public pure returns (uint) {}
}
contract Existing {
    Deployed dc;
    function Existing(address _t) public {
        dc = Deployed(_t);
    }
    function getA() public view returns (uint result) {
        return dc.a();
    }
}
```


information about them can be found. But I do find some clue from Solidity documents:

Expressions that might have a side-effect on memory allocation are allowed, but those that might have a side-effect on other memory objects are not. The built-in functions keccak256, sha256, ripemd160, ecrecover, addmod and mulmod are allowed (even though they do call external contracts).

Check what it says in the bracket. They call external contracts for these built-in functions. Then I just wrote a smart contract with all these functions inside, then I found following mapping for the hard-coded addresses:

```
1 - ecrecover
2 - sha256
3 - ripemd160
4 - sha3
```

For recent version of Solidity compiler, SHA3 has its own opcode, so no external call is needed for this calculation. But you can still see some online smart contracts are using 0x4 to send external call for this functionality.

So far, we have discussed how memory plays an important role in the EVM environment, especially when making external calls to other smart contracts. This is would be the last section for this series. We have talked most of the things you might encounter when you want to analyze the EVM bytecode. Hope it can help you a little bit to understand how it works.

#RESEARCH

AUTHOR

c0zzy

[Read more posts by this author.](#)

ALSO ON TRUSTLOOK BLOG

安全应用程序审核 --
Lionmobi

3 years ago • 1 comment

VirusTotal APK
Malware Detection ...

6 months ago • 1 comment


At Trustlook, we monitor live
feed from VirusTotal (VT). On
a daily basis, we collect ...


VirusTotal
测统计


4 months ago


VirusTotal
旗下产品
扫描服务


What do you think?
6 Responses



Upvote


Funny



Love



Surprised



Angry



Sad


0 Comments

Trustlook blog 

 1 Login ▾

 Favorite

 Tweet

 Share

Sort by Best ▾

Start the discussion...

LOG IN WITH