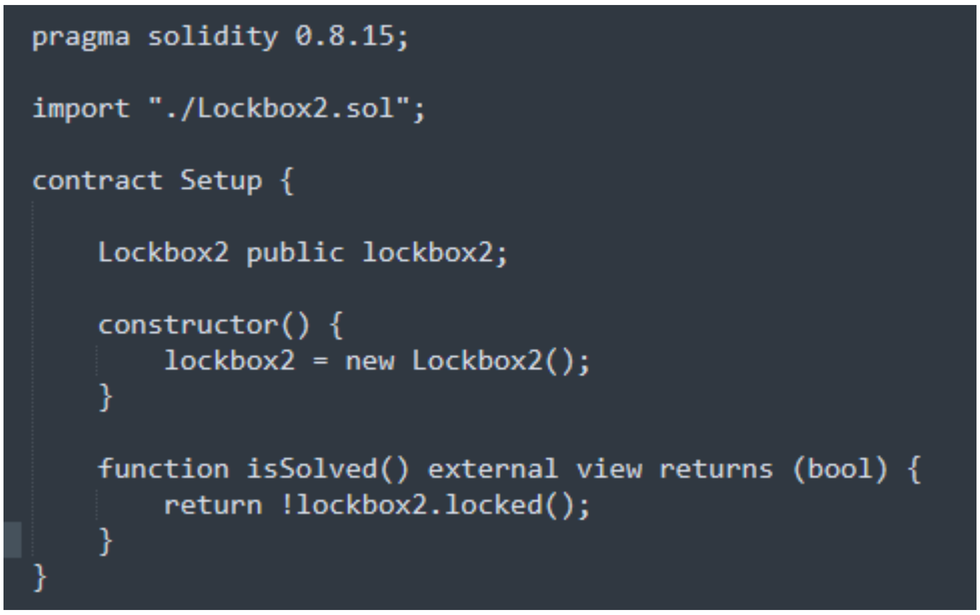
PARADIGM CTF 2022 – LOCKBOX2 WRITEUP

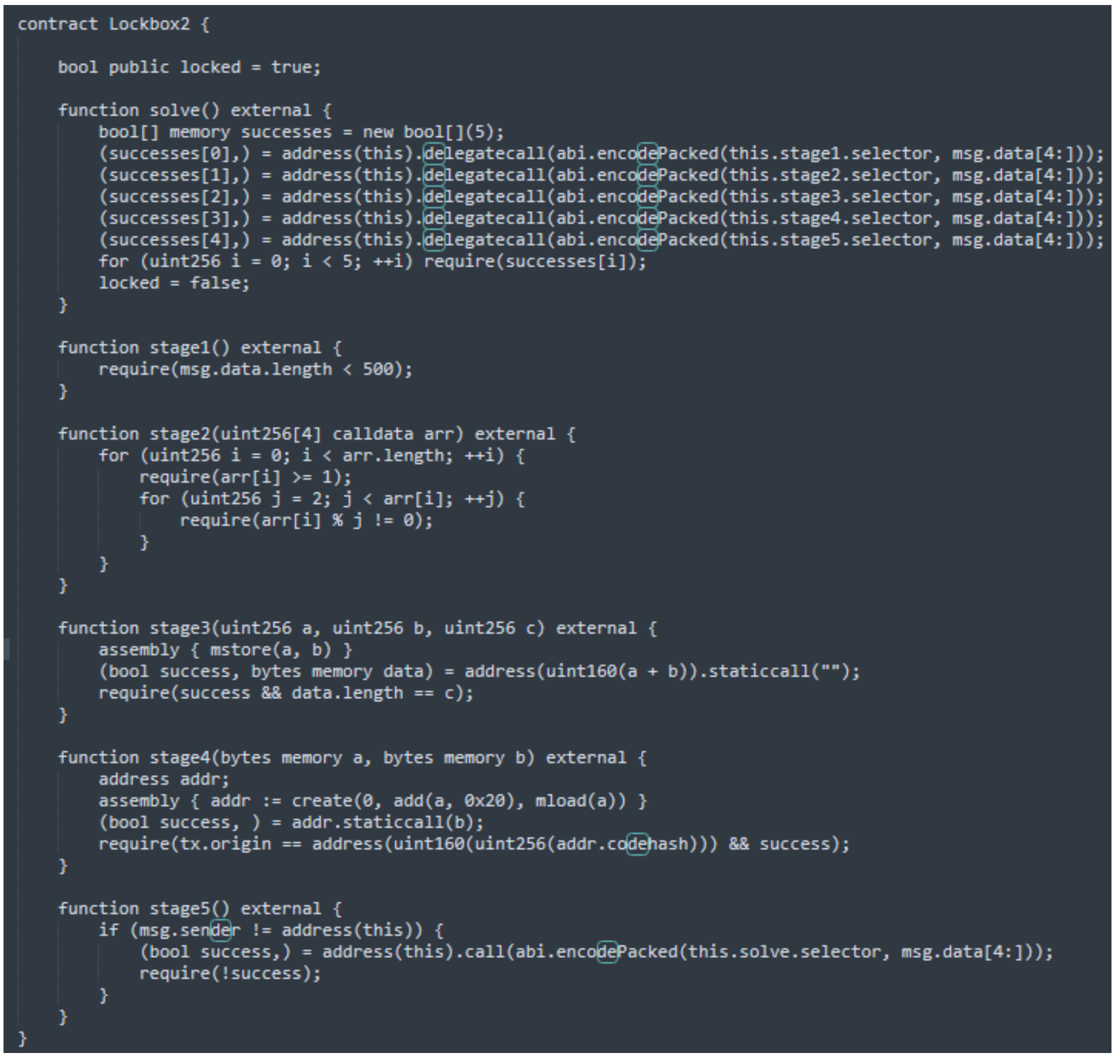
Author: Numen Cyber Technology https://numencyber.com/

Firstly, analyze the *Setup* contract, shown as below:



```
pragma solidity 0.8.15;

import "./Lockbox2.sol";

contract Setup {

    Lockbox2 public lockbox2;

    constructor() {
        lockbox2 = new Lockbox2();
    }

    function isSolved() external view returns (bool) {
        return !lockbox2.locked();
    }
}
```

From above figure can see, it imports *Lockbox2.sol* contract at the beginning, leaving it alone for now. Go to next, it defines *Setup* contract which initializes a new *lockbox2* contract. After then, it is a *isSolved* function. This function is view modified and not on the chain. It returns the non-value of the return value of the *lockbox2* contract of the locked function which is a bool type. So it seems that the premise of capture the flag is to make the *locked* function of *lockbox2* return false.
Next, *Lockbox2* contract：

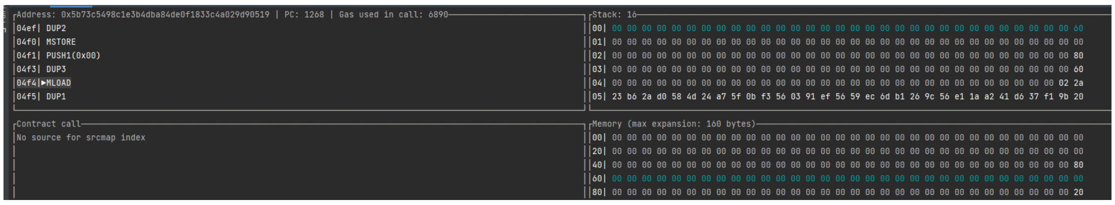


```
contract Lockbox2 {

    bool public locked = true;

    function solve() external {
        bool[] memory successes = new bool[](5);
        (successes[0],) = address(this).delegatecall(abi.encodePacked(this.stage1.selector, msg.data[4:]));
        (successes[1],) = address(this).delegatecall(abi.encodePacked(this.stage2.selector, msg.data[4:]));
        (successes[2],) = address(this).delegatecall(abi.encodePacked(this.stage3.selector, msg.data[4:]));
        (successes[3],) = address(this).delegatecall(abi.encodePacked(this.stage4.selector, msg.data[4:]));
        (successes[4],) = address(this).delegatecall(abi.encodePacked(this.stage5.selector, msg.data[4:]));
        for (uint256 i = 0; i < 5; ++i) require(successes[i]);
        locked = false;
    }

    function stage1() external {
        require(msg.data.length < 500);
    }

    function stage2(uint256[4] calldata arr) external {
        for (uint256 i = 0; i < arr.length; ++i) {
            require(arr[i] >= 1);
            for (uint256 j = 2; j < arr[i]; ++j) {
                require(arr[i] % j != 0);
            }
        }
    }

    function stage3(uint256 a, uint256 b, uint256 c) external {
        assembly { mstore(a, b) }
        (bool success, bytes memory data) = address(uint160(a + b)).staticcall("");
        require(success && data.length == c);
    }

    function stage4(bytes memory a, bytes memory b) external {
        address addr;
        assembly { addr := create(0, add(a, 0x20), mload(a)) }
        (bool success, ) = addr.staticcall(b);
        require(tx.origin == address(uint160(uint256(addr.codehash))) && success);
    }

    function stage5() external {
        if (msg.sender != address(this)) {
            (bool success,) = address(this).call(abi.encodePacked(this.solve.selector, msg.data[4:]));
            require(!success);
        }
    }
}
```

Let’s analyze the code line by line. At first, A global variable-*locked* is defined and initialized as true. Second, it’s *solved* function without parameters. For this function, It first declares a bool type array *successes* with a length of 5. The subscripts from 0 to 4 correspond to the five return values. Continue to look at each line, which is to call the *stage1-5* functions of the current contract, and the calldata data starts from the 4th bit of *msg.data*. Starting from the fourth digit, that is, not counting the function signature. Then assign the success of each call as a bool value to the *successes* array. Then there is a loop through the array of bool type, if each is true, continue to run, set *locked* to false. Only the solve entry can change the *locked* variable. It seems that the key problem is the 5 calls of the *solve* function.

Continue to analyze the following 5 calls.
Stage1，it requires the length of *msg.data* less than 500.
Stage2，The parameter passed in is a uint256 array with length 4. In the first 4 lines of incoming *msg.data*, a line of 32 bytes, each line represents a number. And then all 4 numbers must be satisfied the conditions that it cannot divide any number except 1 and itself. This is easy to be constructed.

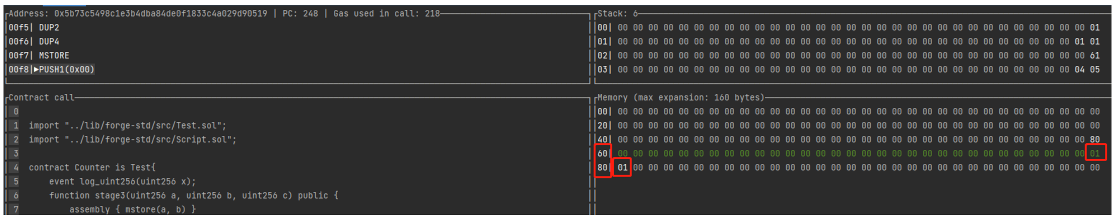
Stage3，The incoming parameters are 3 uint256 type data. First, *mstore* is called, and *b* is stored in the position of a in the stack, *b* represents the data, and *a* represents the position in the stack. Then a program call, where the sum of a, b represents an address, and then the *staticcall* function statically calls this address. We cannot construct such an address, so the return must be empty.
Next line, it requires the length of the return value needs to be equal to the incoming parameter *c*. According to *stage2*, it is known that *c* must not be able to divide any number except 1 and itself, and the length of the return value here must be 0, which seems impossible. Because I noticed that there is an *mstore* at the beginning, which is stored anywhere in memory, this *mstore* must be meaningful. First of all, through the data, we learned about the characteristics of *solidity*. Since the return value of the call is empty, it will be stored in a special location 0x60 (https://docs.soliditylang.org/en/latest/internals/layout_in_memory.html#layout-in-memory) Next, we debug to verify.



```
Address: 0x5b73c5498c1e3b4dba84de0f1833c4a029d90519 | PC: 1268 | Gas used in call: 6890
04ef| DUP2
04f0| MSTORE
04f1| PUSH1(0x00)
04f3| DUP3
04f4|►MLOAD
04f5| DUP1

Contract call
No source for srcmap index
```

```
Stack: 16
00| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60
01| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80
03| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60
04| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 2a
05| 23 b6 2a d0 58 4d 24 a7 5f 0b f3 56 03 91 ef 56 59 ec 6d b1 26 9c 56 e1 1a a2 41 d6 37 f1 9b 20

Memory (max expansion: 160 bytes)
00| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80
60| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 20
```

It can be seen that after the static call, *mload* reads the value at the 0x60 position in the stack, so we only need to store data in the specified position in *mstore*, and then let stage3 pass. We can pass in 0x61, 0x0101, 0x1.
First save 0101 to 0x61 through *mstore*. Because *mstore* is used, 32 bytes of data are stored, 0x0101 high-order bits are filled with 0, and 32 bytes are filled. Because the 60 position has a total of 32 bytes, but *mstore* starts from the 61 position, and the entire 60 can not be stored, so 80 positions will be occupied, shown as below figure.

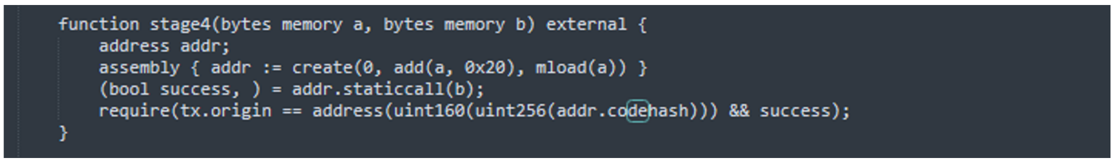


```
Address: 0x5b73c5498c1e3b4dba84de0f1833c4a029d90519 | PC: 248 | Gas used in call: 218
00f5| DUP2
00f6| DUP4
00f7| MSTORE
00f8|►PUSH1(0x00)

Contract call

import "../lib/forge-std/src/Test.sol";
import "../lib/forge-std/src/Script.sol";

contract Counter is Test{
    event log_uint256(uint256 x);
    function stage3(uint256 a, uint256 b, uint256 c) public {
        assembly { mstore(a, b) }
```

```
Stack: 6
00| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
01| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 01
02| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 61
03| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04 05

Memory (max expansion: 160 bytes)
00| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80
60| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
80| 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

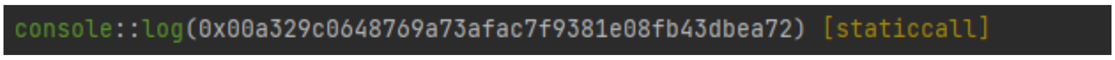
Now that the 60 position is successfully controlled, *mstore(a,b)* will fetch the number at 0x60 due to it is before the declaration of bytes memory data and returned data is empty. So *mstore* is used to control the input parameters to pass stage3.

Stage4，stage4 passes in two parameters, both of them are bytes type. Firstly, a contract is created with parameter *a* and saved on the chain. It will return an address. Then perform a static call with parameter *b*. the following code is a *require* judgment which requires the code of the newly generated contract must equals to the public key of *tx.origin*. This is also easier to construct。

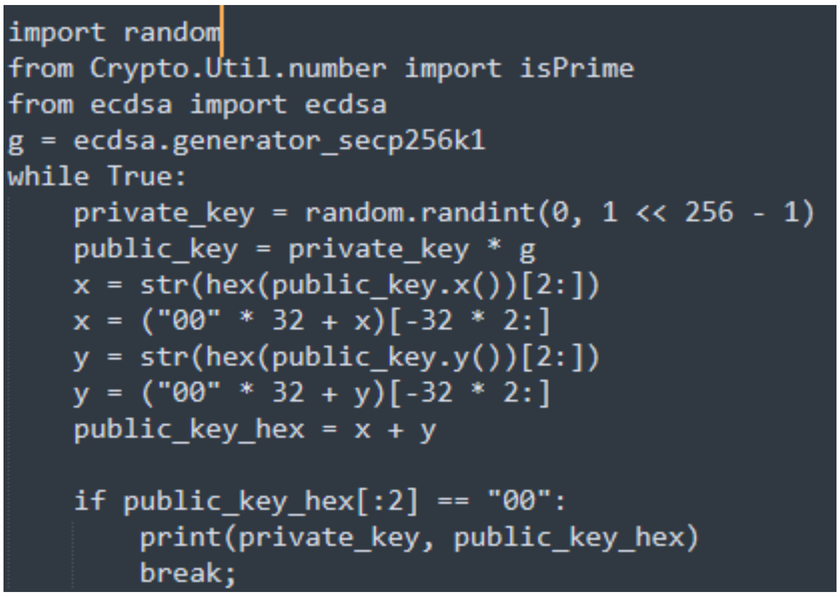


```
function stage4(bytes memory a, bytes memory b) external {
    address addr;
    assembly { addr := create(0, add(a, 0x20), mload(a)) }
    (bool success, ) = addr.staticcall(b);
    require(tx.origin == address(uint160(uint256(addr.codehash))) && success);
}
```

Next we print *tx.origin*， and found there are two more 0s in front of *tx.origin*. That means when constructing the public key, the first two digits are required to be '00'.

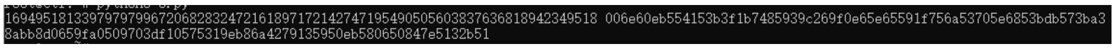


```
console::log(0x00a329c0648769a73afac7f9381e08fb43dbea72) [staticcall]
```



```
import random
from Crypto.Util.number import isPrime
from ecdsa import ecdsa
g = ecdsa.generator_secp256k1
while True:
    private_key = random.randint(0, 1 << 256 - 1)
    public_key = private_key * g
    x = str(hex(public_key.x())[2:])
    x = ("00" * 32 + x)[-32 * 2:]
    y = str(hex(public_key.y())[2:])
    y = ("00" * 32 + y)[-32 * 2:]
    public_key_hex = x + y

    if public_key_hex[:2] == "00":
        print(private_key, public_key_hex)
        break;
```

We first use the above script to run a public key and a private key, and output a public key whose first two digits are 0.

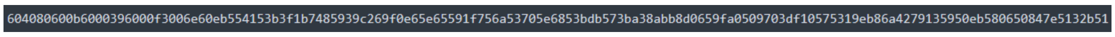


```
16949518133979797996720682832472161897172142747195490505603837636818942349518 006e60eb554153b3f1b7485939c269f0e65e65591f756a53705e6853bdb573ba3
8abb8d0659fa0509703df10575319eb86a4279135950eb580650847e5132b51
```

Then we need to construct an *opcode* to keep the public key content on the chain.

PUSH1 0x40
DUP1
PUSH1 0x06
PUSH1 0x0
CODECOPY
PUSH1 0
RETURN

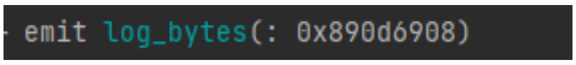
Concatenate the above opcode with the constructed public key. First explain the meaning of *opcode*, push 40 into the stack, copy it, and then push both 06 and 0 into the stack. At this point, the contents of the stack are 0, 0b, 40, and 40 from the top of the stack down. *codecopy* receives 3 parameters, 0, 0b, 40 from the top of the stack which represent the target position, the current position and the code length respectively. The code of public key is 64 bytes, so the length is 0x40. Because the content of the public key is output of the *opcode*, so the current position of the public key is the sum of the length of the opcode bytecode, that is 11=0x0b. Afterwards, we need to copy the public key (code) from 0b to 0. Put 0 on the stack, and now there are 0, 40 in the stack. Then execute *return* command, and return the content from position 0 to 40. So far, the *opcode* deploys the content of the public key to the chain. Full version likes below:



```
604080600b6000396000f3006e60eb554153b3f1b7485939c269f0e65e65591f756a53705e6853bdb573ba38abb8d0659fa0509703df10575319eb86a4279135950eb580650847e5132b51
```

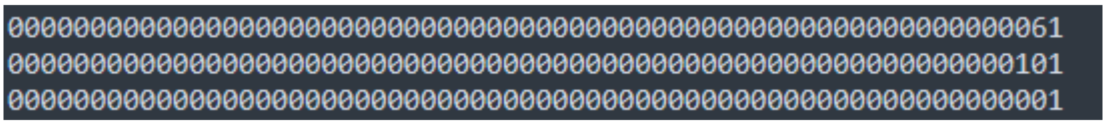
That means, *a* should be passed in the bytes array, *b* shall be passed in 0x0.

Stage5, the last one. It calls back the *solve* function of the current contract. But if it fails to return, the first pass succeeds and the second pass fails. We control the *gas* when we consider the call. This is done later when debugging, which is easier to handle. First construct the data. First look at the function signature of *solve*.



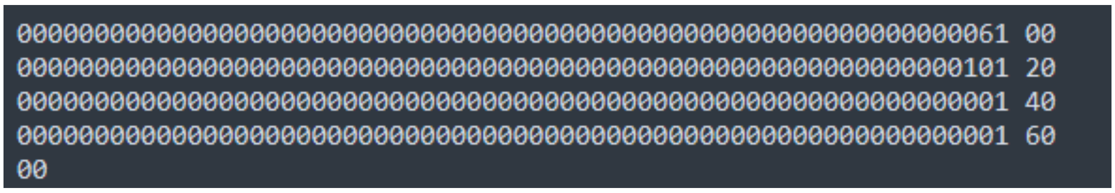
```
emit log_bytes(: 0x890d6908)
```

Then it is to construct the data. First, you need to get 4 uint256 type data, which satisfies >=1, and can only be divided by 1 and itself. The above has been constructed.



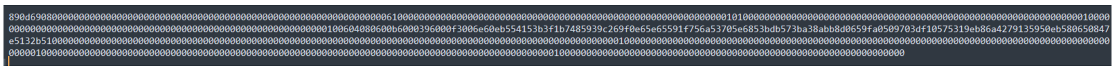
```
0000000000000000000000000000000000000000000000000000000000000061
0000000000000000000000000000000000000000000000000000000000000101
0000000000000000000000000000000000000000000000000000000000000001
```

This structure can meet stage2, stage3, the reasons have been explained above.
Next stage4. Because there are 3 integers only ， missing one, stage 2 is not satisfied, so it has to find another number.
First of all, let's find out how the bytes string is stored in memory. Although stage4 lets you pass 2 strings, but only *a* is useful. For string value， it stores the length first, and then store the content. The *msg.data* in the first line represents the offset, so our next construction goes to line 4, which just conflicts with the fourth parameter of stage2. Because it starts from the 61th position, the 4th line is definitely not full, we can add 0 to the following data. So we construct data with the length of 0100, and let its performance in the 4th line be 01, which satisfies stage2.

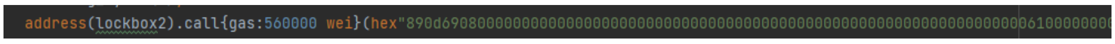


```
0000000000000000000000000000000000000000000000000000000000000061 00
0000000000000000000000000000000000000000000000000000000000000101 20
0000000000000000000000000000000000000000000000000000000000000001 40
0000000000000000000000000000000000000000000000000000000000000001 60
00
```

We only need to construct as above to satisfy stage1, 2, and 3. The data required by Stage 4, it can be done by *abi* encoding. Then fill it with 0 until the length up to 0100 bytes. The full data shown as below:

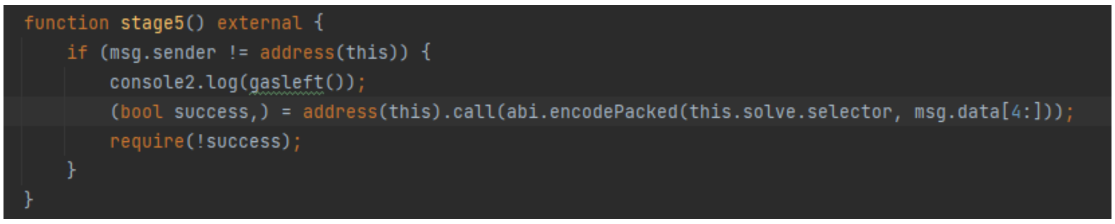


```
890d6908000000000000000000000000000000000000000000000000000000000000006100000000000000000000000000000000000000000000000000000000000001010000000000000000000000000000000000000000000000000000000000000010000
000000000000000000000000000000000000000000000000000000000100604080600b6000396000f3006e60eb554153b3f1b7485939c269f0e65e65591f756a53705e6853bdb573ba38abb8d0659fa0509703df10575319eb86a4279135950eb580650847
e5132b51000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000001000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010000000000000000000000000000000000000000000000000000000000000000000
```



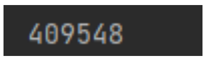
```
address(lockbox2).call{gas:560000 wei}(hex"890d690800000000000000000000000000000000000000000000000000000000000000610000000
```

Controlled by contract call, and passing in our *msg.data*. Now we only need to find the gas cost of stage5. Shown as below figure:

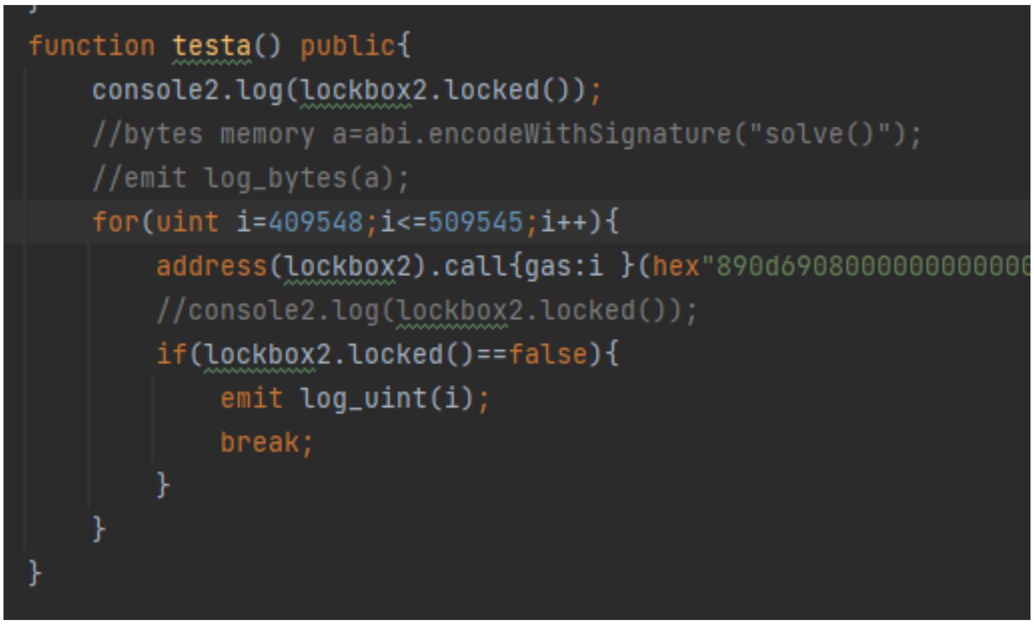


```
function stage5() external {
    if (msg.sender != address(this)) {
        console2.log(gasleft());
        (bool success,) = address(this).call(abi.encodePacked(this.solve.selector, msg.data[4:]));
        require(!success);
    }
}
```

We print the result of gas here directly, the result likes below :

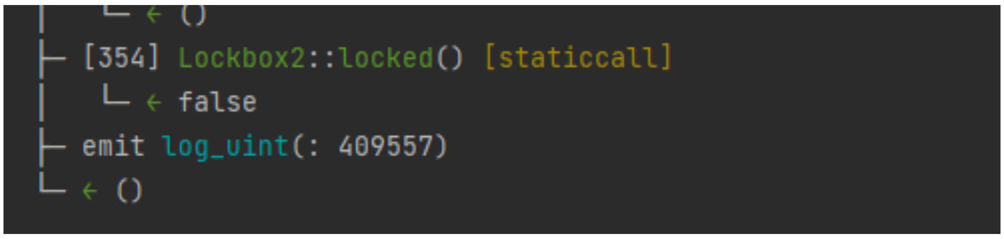


```
409548
```

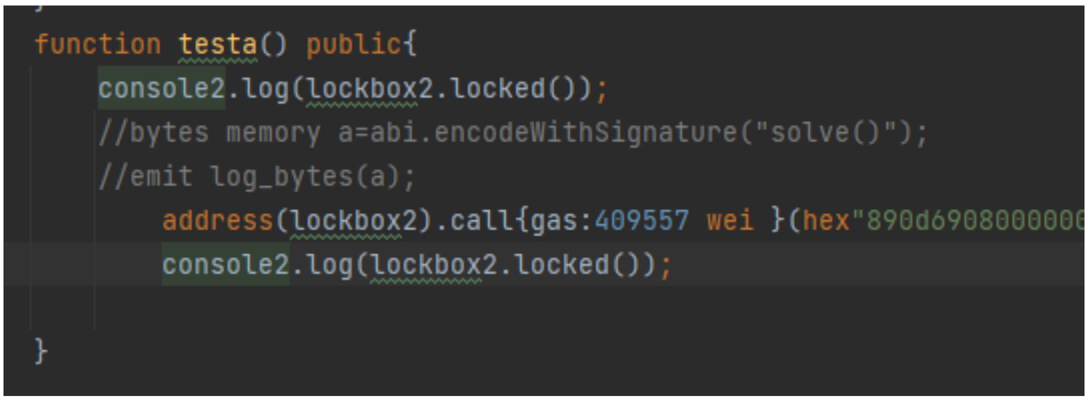


```
function testa() public{
    console2.log(lockbox2.locked());
    //bytes memory a=abi.encodeWithSignature("solve()");
    //emit log_bytes(a);
    for(uint i=409548;i<=509545;i++){
        address(lockbox2).call{gas:i }(hex"890d690800000000000
        //console2.log(lockbox2.locked());
        if(lockbox2.locked()==false){
            emit log_uint(i);
            break;
        }
    }
}
```

Since it I a bit troublesome to find, it’s better to engage in directly, like below:

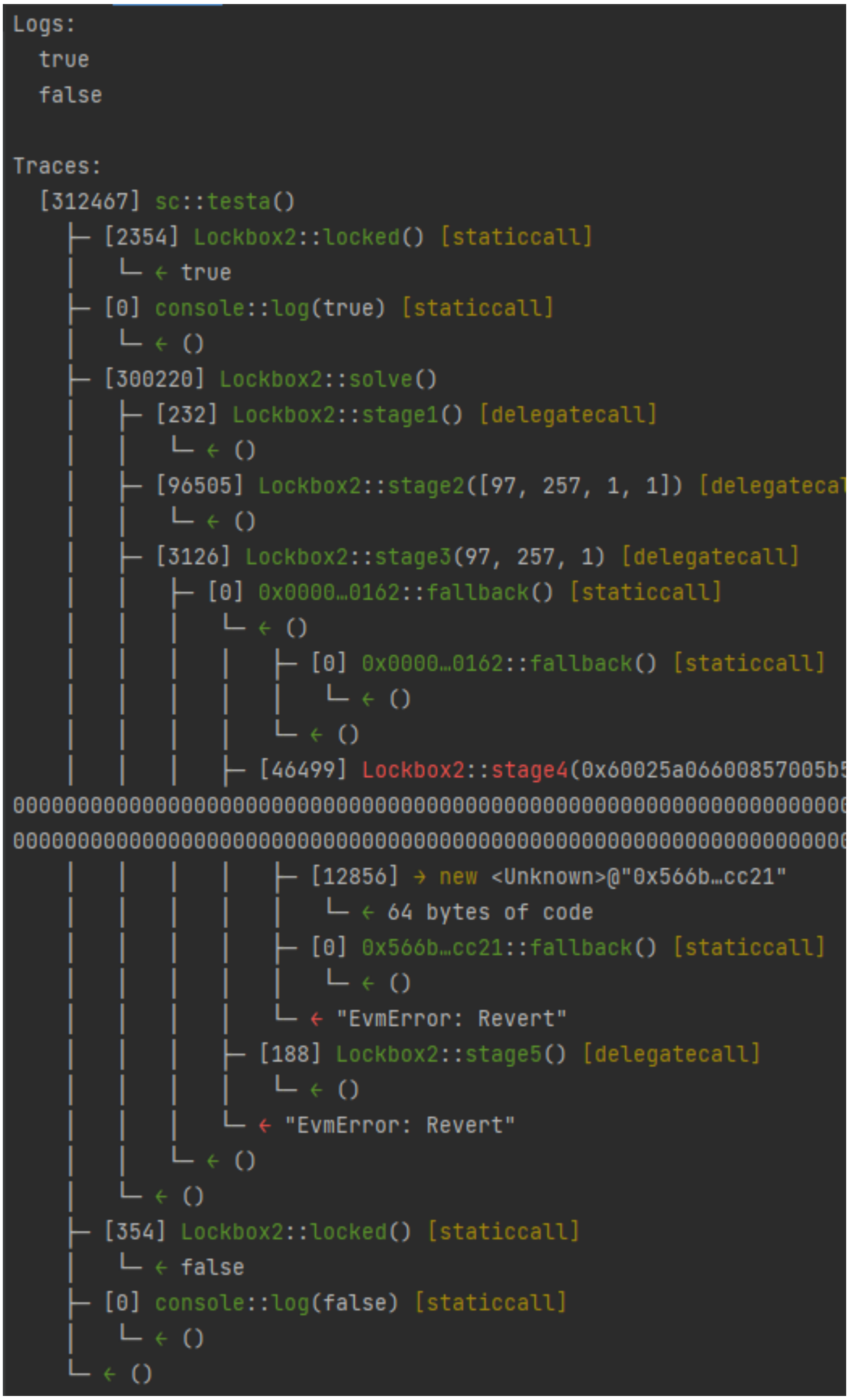


```
|   └─ ← ()
├─ [354] Lockbox2::locked() [staticcall]
|   └─ ← false
├─ emit log_uint(: 409557)
└─ ← ()
```



```
function testa() public{
    console2.log(lockbox2.locked());
    //bytes memory a=abi.encodeWithSignature("solve()");
    //emit log_bytes(a);
        address(lockbox2).call{gas:409557 wei }(hex"890d69080000000
        console2.log(lockbox2.locked());

}
```

Then we see that the result of *locked* is false after executing *solve*. Capturing the flag.



```
Logs:
  true
  false

Traces:
  [312467] sc::testa()
    ├─ [2354] Lockbox2::locked() [staticcall]
    │   └─ ← true
    ├─ [0] console::log(true) [staticcall]
    │   └─ ← ()
    ├─ [300220] Lockbox2::solve()
    │   ├─ [232] Lockbox2::stage1() [delegatecall]
    │   │   └─ ← ()
    │   ├─ [96505] Lockbox2::stage2([97, 257, 1, 1]) [delegateca
    │   │   └─ ← ()
    │   ├─ [3126] Lockbox2::stage3(97, 257, 1) [delegatecall]
    │   │   ├─ [0] 0x0000…0162::fallback() [staticcall]
    │   │   │   └─ ← ()
    │   │   │   │   ├─ [0] 0x0000…0162::fallback() [staticcall]
    │   │   │   │   │   └─ ← ()
    │   │   │   │   └─ ← ()
    │   │   │   ├─ [46499] Lockbox2::stage4(0x60025a06600857005b5
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
    │   │   │   │   ├─ [12856] → new <Unknown>@"0x566b…cc21"
    │   │   │   │   │   └─ ← 64 bytes of code
    │   │   │   │   ├─ [0] 0x566b…cc21::fallback() [staticcall]
    │   │   │   │   │   └─ ← ()
    │   │   │   │   └─ ← "EvmError: Revert"
    │   │   │   ├─ [188] Lockbox2::stage5() [delegatecall]
    │   │   │   │   └─ ← ()
    │   │   │   └─ ← "EvmError: Revert"
    │   │   └─ ← ()
    │   └─ ← ()
    ├─ [354] Lockbox2::locked() [staticcall]
    │   └─ ← false
    ├─ [0] console::log(false) [staticcall]
    │   └─ ← ()
    └─ ← ()
```