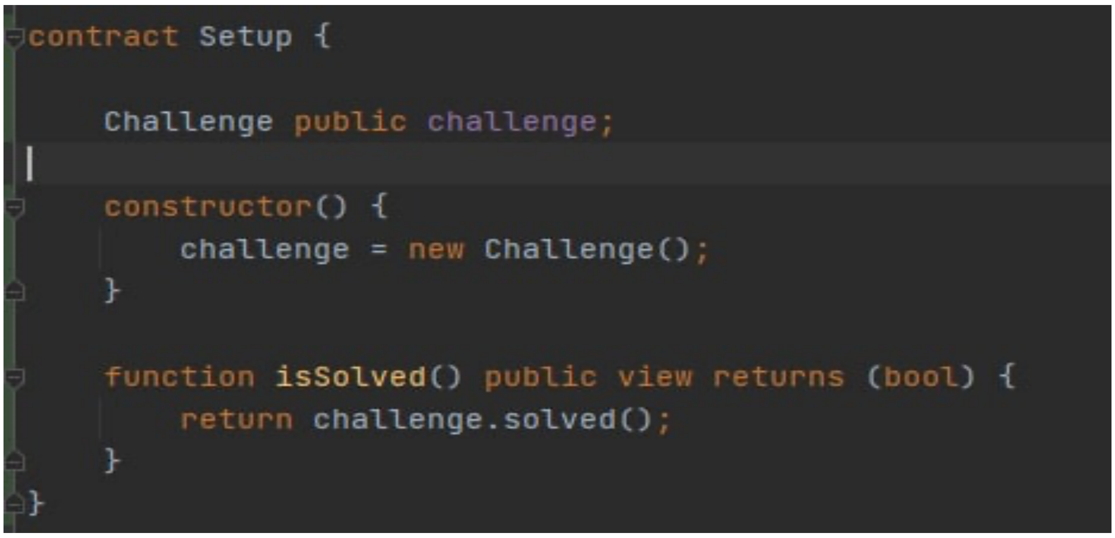
PARADIGM CTF 2022 – SOURCE CODE WRITEUP

Following the previous article about rescue analysis, today, let’ s discuss the **source code** analysis together.

Code Analysis

Firstly, let’ s analyze the Setup contract, as shown in figure 1. From the code, we can see that the premise of getting the flag is isSolved returns true. Next to analyze the challenge contract. The challenge contract mainly has two functions- solve and safe, as shown in figure 2.



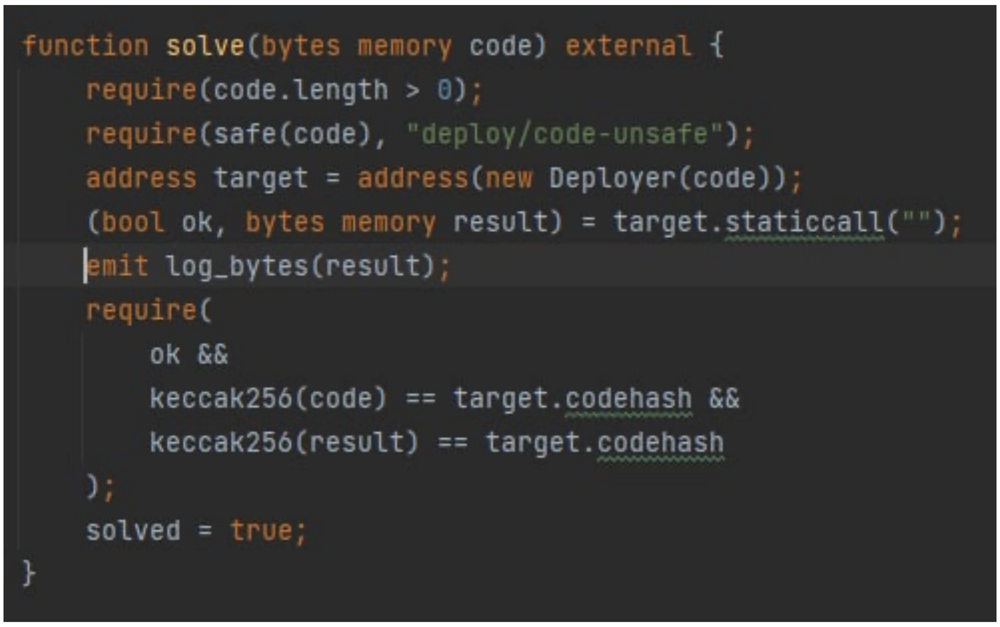
```
contract Setup {

    Challenge public challenge;

    constructor() {
        challenge = new Challenge();
    }

    function isSolved() public view returns (bool) {
        return challenge.solved();
    }
}
```

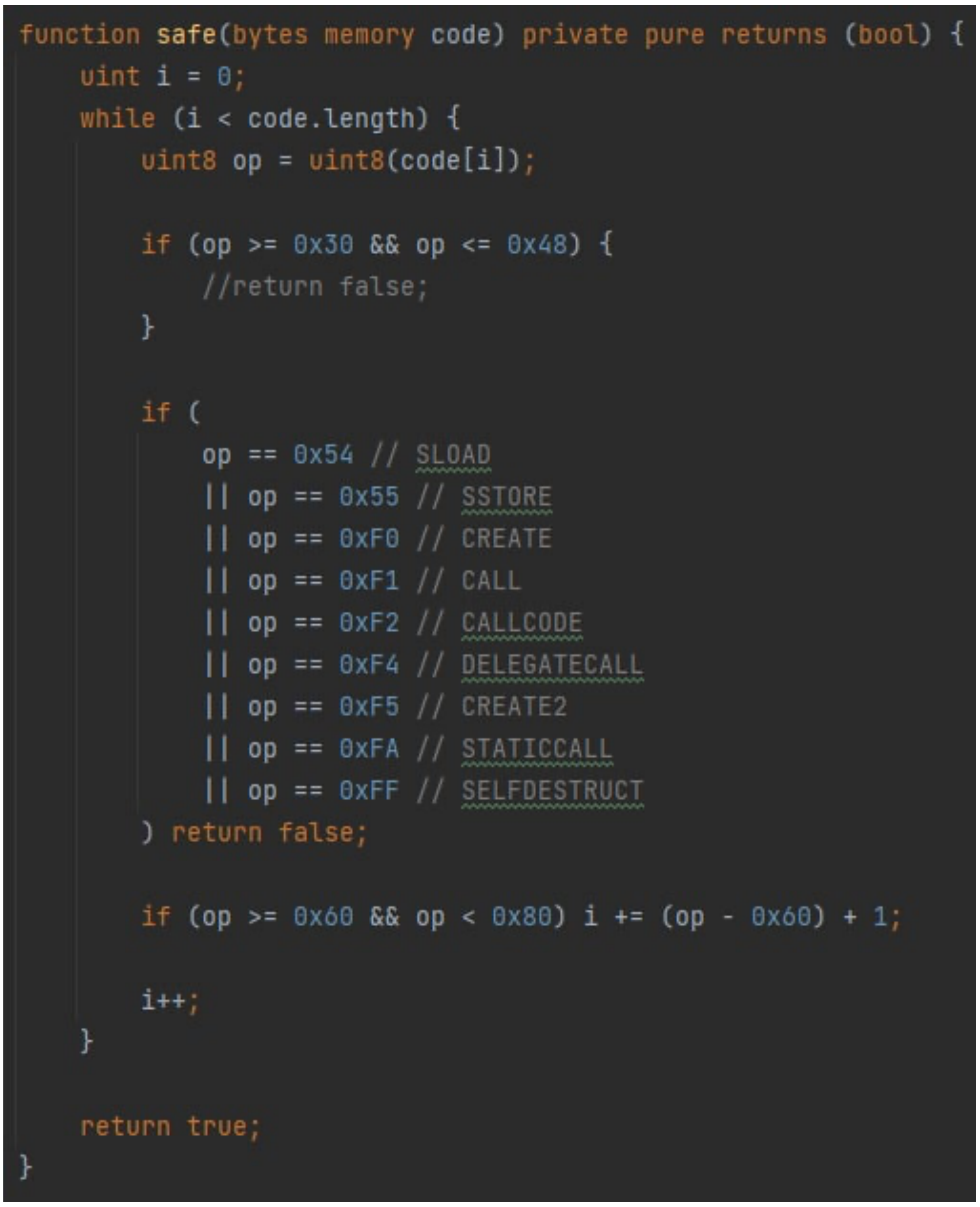
For solve function, if we want the isSolved function of the Setup contract to return true, we must change the variable solved. The only way to change the solved is to call the solve function and continue to analyze the solve function. The solve function passes in a variable of type bytes, first checks that the length is not 0, and then calls the safe function. We can see that the parameter of the safe function is the code passed in by the solve function, and then the safe function returns a bool value. If we want to continue running, we must make the safe function return true. For the safe function, let’ s leave it to be analyzed later on, we continue to look down. The next line of code uses the incoming code as a parameter to directly create a new contract. When creating a new contract, it will call the initialization function -constructor function. After that, staticcall, the new contract, returns two values. One is a bool which indicates whether the call is successful or not. The other one is a bytes type value which indicates the return data. The last judgment statement requires that the staticcall is called successfully and the returned data (no matter how the call evm, the return is always the same result) is consistent with the bytecode of the contract.



```
function solve(bytes memory code) external {
    require(code.length > 0);
    require(safe(code), "deploy/code-unsafe");
    address target = address(new Deployer(code));
    (bool ok, bytes memory result) = target.staticcall("");
    emit log_bytes(result);
    require(
        ok &&
        keccak256(code) == target.codehash &&
        keccak256(result) == target.codehash
    );
    solved = true;
}
```

Figure 2

After the above judgment is passed, the solved value becomes true. Seeing this , we found there are many methods to make the bytecode of the contract the same as the return value of the call. But it shouldn't be so simple, because there's a key safe function in the middle.



```
function safe(bytes memory code) private pure returns (bool) {
    uint i = 0;
    while (i < code.length) {
        uint8 op = uint8(code[i]);

        if (op >= 0x30 && op <= 0x48) {
            //return false;
        }

        if (
            op == 0x54 // SLOAD
            || op == 0x55 // SSTORE
            || op == 0xF0 // CREATE
            || op == 0xF1 // CALL
            || op == 0xF2 // CALLCODE
            || op == 0xF4 // DELEGATECALL
            || op == 0xF5 // CREATE2
            || op == 0xFA // STATICCALL
            || op == 0xFF // SELFDESTRUCT
        ) return false;

        if (op >= 0x60 && op < 0x80) i += (op - 0x60) + 1;

        i++;
    }

    return true;
}
```

Figure 3

Then let’ s come back to the safe logic. The safe function disables some opcodes by traversing the incoming bytes-type string, and returns false directly if a disabled opcode is matched.

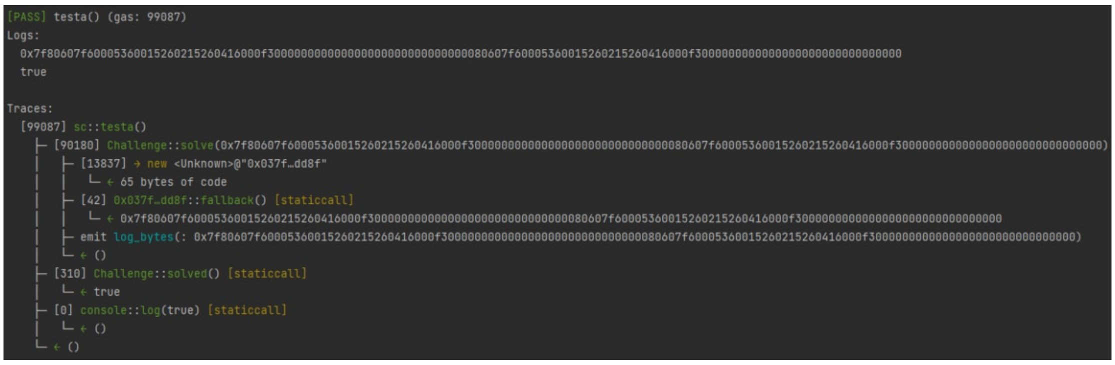
If we want the safe function to return true, the opcode we pass in must be in the "whitelist" defined by the safe function. After that, using code as a parameter, new a contract, the last judgment requires that the bytecode of the contract and the return result of an arbitrary call to this contract is the same no matter how called the function. After completing all of the above require judgments, you can change solved and get flag. It requires the sourcecode of the contract is the same as the return value of the contract call.

In fact, we have many ways to meet this condition, such as **CODECOPY, CALLCODE, CALL** which can achieve contract code and call return consistency. But these opcodes are in the "blacklist" of the safe contract. So, need to think of other methods. The general idea is that we try to construct a series of operation execution instructions, and then we push this data, and then execute this data instruction, the instruction content of this data is to use **DUP1, MSTORE, RETURN** these to manipulate the contents of the stack, and return (we have taken a series of operation instructions as parameters into the stack in the first step), the key operation is **DUP1**. You need to put the operation instruction as a parameter and push once, and the code needs to be executed once. So the return of the call appears twice, then need to copy it once with **DUP1**. For **MSTORE** type directives, there are only **MSTORE 8** and **MSTORE**. One is to save 1 byte, the other one is to save 32 bytes. 1 byte is definitely not enough, 32 may not be used up, we can supplement 0 later (using **STOP**). So the length of the push instruction must be 32 bytes at the beginning, if the **MSTORE** storage will be in the high bit after the 0, then it will definitely affect the return value of **RETURN**.

Final construction:

```
PUSH32
0x80607f60005360015260215260416000f3000000000000000000000000000000
DUP1
PUSH1 0x7f
PUSH1 0x00
MSTORE8
PUSH1 0x01
MSTORE
PUSH1 0x21
MSTORE
PUSH1 0x41
PUSH1 0x00
RETURN
STOP
STOP
STOP
STOP
STOP
STOP
STOP
STOP
STOP
STOP
STOP
STOP
STOP
STOP
STOP
```

Putting a 32-byte content into the stack and copy it once (take the data copy at the top of the stack), so the 2 elements in the stack are the content of the instruction data. **DUP1** is critical, we need to copy the content of the instruction data for output. The initial instruction **PUSH32** corresponds to bytecode is 7f, so 7f is constructed. Put 7f into the stack, save the 00 position, occupy 1 byte, can only use **MSTORE8**. As mentioned earlier, if you use **MSTORE**, it will make up 0 at a high level, make up 32 bits, and affect **RETURN** value. The rest are operations in the instruction data. At this time, there are two elements in the stack that are the contents of the instruction data, and the top of the stack is stored at position 01, one data occupies 32 bytes, and then there is 1 left, and this is stored at position 21 (32 bytes, so 01 + 20 = 21, 16). Finally **RETURN** returns from 0, a 1-byte data and two 32-byte data, a total length of 0x41, **STOP** corresponds to 0. Therefore, the content of one 7f and two instruction data are returned, which is consistent with the content of the code (code: **pushed** a 32-byte data, hard-coded the instruction data as a parameter, and then executed the content of the instruction data, so it is also the content of one 7f and two instruction data).



```
[PASS] testa() (gas: 99087)
Logs:
  0x7f80607f60005360015260215260416000f300000000000000000000000000000080607f60005360015260215260416000f3000000000000000000000000000000
  true

Traces:
  [99087] sc::testa()
    ├─ [90180] Challenge::solve(0x7f80607f60005360015260215260416000f300000000000000000000000000000080607f60005360015260215260416000f3000000000000000000000000000000)
    │   ├─ [13837] → new <Unknown>@"0x037f…dd8f"
    │   │   └─ ← 65 bytes of code
    │   ├─ [42] 0x037f…dd8f::fallback() [staticcall]
    │   │   └─ ← 0x7f80607f60005360015260215260416000f300000000000000000000000000000080607f60005360015260215260416000f3000000000000000000000000000000
    │   ├─ emit log_bytes(: 0x7f80607f60005360015260215260416000f300000000000000000000000000000080607f60005360015260215260416000f3000000000000000000000000000000)
    │   └─ ← ()
    ├─ [310] Challenge::solved() [staticcall]
    │   └─ ← true
    ├─ [0] console::log(true) [staticcall]
    │   └─ ← ()
    └─ ← ()
```

Summary

Flexible use of **dup**, the key point of this problem is **dup**, this operation code has **dup1-16**. Respectively, the number starts at the top of the stack and copies the elements in the current position. It is precisely by using the **dup** operation that the **push** operation is omitted, so that the bytecode of the contract can be returned in the same way as the call of the contract.

Numen Cyber Technology https://numencyber.com/