

# iCSC 2018 Workshop

---

Title: Computing for Decentralized Systems.

Domain: Distributed Systems, Decentralization, Consensus, Byzantine Fault Tolerance, Blockchain, Smart Contracts.

Speaker: Alejandro Avilés (@OmeGak)

Date: 08/03/2018

License: Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

## Capture The Flag

---

The workshop will be done around [Ethernaut](#), a capture-the-flag game by Zeppelin Solutions. It teaches, or gives leads, to the basics of Solidity, web3, and smart contracts in Ethereum.

The ethernaut is a Web3/Solidity based wargame inspired on overthewire.org and the Ethernauta comic, played in the Ethereum Virtual Machine. Each level is a smart contract that needs to be 'hacked' in order to advance.

## Set up

Before we take off, let's set up everything we will need to go through the CTF.

Software requirements:

- Google Chrome or Firefox
- [Metamask extension](#)

Metamask configuration:

- Select `Ropsten test network`.
- Create a wallet.

Collecting funds:

- Go to <https://faucet.metamask.io/>
- Request a bunch of Ether

## What's going on?

The Metamask extension enables the browser to interact with Ethereum networks by speaking

to trusted Ethereum nodes hosted by Metamask.

This capture the flag challenge will require us to send transactions that will call functions in smart contracts. Both getting transactions included in blocks and running smart contract code costs fee (called gas in Ethereum), and so we need to get some Ether first.

We don't want to spend real money in this capture the flag challenge, so we are doing everything in one of the test networks, named Ropsten. It's common to have faucets, contracts that send Ether for free, in test networks so that people can easily get some funds to play with and test stuff.

Each level in Ethernaut is a smart contract deployed exclusively for each player. To hack each one of these smart contracts we will need to make calls to their functions and send transactions to them. This is possible directly from the browser's console using the web3 Javascript library, automatically imported by the Metamask extension. The Ethernaut website also adds some handy shortcuts.

Finally, the web3 library will be used to read and write from/to the blockchain. Reading happens instantly and for free from the Ethereum node the Metamask extension is connected to. Writing, on the other hand, implies changing the state of the blockchain and we will be prompted to confirm a transaction before the Metamask extension broadcasts it.

## Level 0

This level is basically teaching how to interact with a contract by calling different functions from the console starting with `contract.info()`.

Let's first look at what an smart contract ABI is. If we inspect the master contract that spawns level contracts, [Ethernaut.sol](#) we can see it has some functions. We can directly read the signatures of the contract functions directly from the console `ethernaut.abi`.

To spawn a level contract we need to click on the blue button at the bottom of the page. This will send a transaction to the `ethernaut` contract and we will have our level contract in `contract` once the transaction gets confirmed.

Hints:

- Call `contract.abi` to figure out password

Solution:

```
await contract.info()
```

```
contract.abi
password = await contract.password()
await contract.authenticate(password)
```

## Level 1

In this level the player is required to take ownership of the contract and reduce its balance to 0. Taking the ownership of the contract is basically changing the member `owner` from `level` to `player`, so that the player can deplete the funds calling `withdraw()`.

The challenge in this level is to figure out how to send Ether in function calls. This would usually require to dig into the Solidity documentation, but it's really a mess. It's better to go through the code step by step, giving the hint once anybody asks the right question.

Explanations:

- `msg.value` is the amount of funds (Ether) sent to the contract in the transaction.
- `msg.value` can be set in any call as the last argument: `fn(..., {value: x})`.
- The fallback function can be called with `sendTransaction(...)`.

Hints:

- Where can we change ownership?
- `contribute()` needs to receive value, so we need to find the way to do it.

Solution:

```
await contract.contribute({value: 1})
await contract.sendTransaction({value: 1})
await contract.withdraw()
await contract.owner() == player
```

## Level 2

In this level the player is also required to take ownership of the contract. The only place where this can be done is in the, at first sight, constructor. The constructor is the function that gets called when creating the contract, with the same name as the contract, and it sets `owner` to `level`.

Everything looks alright, until we realize the constructor is not the constructor because it's misspelled. `Fallout` is not the same as `Fal1out`. Because that function is public anybody

can call it and become the owner of the contract.

Hints:

- The constructor is the function that is called exactly the same as the contract.

Solution:

```
await contract.Fallout()  
await contract.owner() == player
```

## Level 3

This level presents a token contract with a supply limit. The player is given 20 tokens and the task is to increase that number, but by looking at the functions available in the contract there doesn't seem to be a direct way to do it.

Here the purpose is producing an integer overflow in the amount of tokens assigned to the player. `balances` is a mapping of addresses to unsigned integers used to keep track of people's tokens. The problem is within the logic of `transfer()`, which allows the sender to transfer tokens to another address by first deducting them. This means that if more tokens than available to him are transferred, the `uint` will overflow and become a very high number.

Hints:

- Pay attention to the types.
- What happens to an unsigned integer if it goes below zero?
- Who could you send it to so that the value remains "below" zero?

Solution:

```
myBalance = await contract.balanceOf(player)  
myBalance = myBalance.c[0]  
await contract.transfer(level, myBalance + 1)  
myNewBalance = await contract.balanceOf(player)  
myNewBalance.c
```

## Level 4

In this level the player is presented with two contracts and is expected of taking ownership of

the second one, `Delegation`. This contract is correctly constructed, so the player cannot use the same trick as before.

The strategy for this one is exploiting a vulnerability of `delegatecall`. `delegatecall` allows calling a function from another contract, `Delegate` in this case, without changing the scope. In practice, this means the contract performing the `delegatecall`, the callee, is trusting the called contract to mess with the state of the callee. The function `pwn()` in `Delegate` changes the owner and if called from `Delegation` it will allow the player to become the owner.

`delegatecall` is a low-level builtin function in Solidity and takes as argument a string of bytes that is expected to represent the SHA3 of the signature of the function to be called. This can be easily done with the `web3` library.

Explanations:

- Explain `delegatecall`.
- Explain how to send data to fallback function with `sendTransaction({data: ...})`.
- Explain that SHA3 is available through `web3`.

Hints:

- Which contract do you think you need to take ownership of?

Solution:

```
await contract.sendTransaction({data: web3.sha3("pwn()")})
await contract.owner() == player
```

## Level 5

This level introduces an empty contract and the task is to make it's balance go up. Because it doesn't have any payable function it's impossible to send Ether.

To hack this contract we need to be aware of the existence of the `selfdestruct(...)` builtin function in Solidity. Any contract can call this function and all its funds will be transferred to the address passed to it. There is no way for a contract to reject this transaction.

This will require us to deploy a smart contract that will self-destruct to send funds to the level contract that we need to break. This can be done using [Remix](#), a web-based IDE, to code, compile, and deploy smart contracts. It's important to select the `Injected web3` environment to deploy the contract in the same Ethereum network as we are playing.

## Explanations:

- `selfdestruct(...)` and how it can send funds to any address.
- How a smart contract can be written and deployed from [Remix](#)

## Solution:

```
await getBalance(contract.address)
contract.address
```

```
contract Samaritan {
  function Samaritan() public payable {
    require(msg.value > 0);
  }

  function help() public {
    selfdestruct(...);
  }
}
```

- Replace `...` with `contract.address`
- Deploy contract with some Ether on it.
- Call `help`.

```
await getBalance(contract.address)
```