

Snitch | ECW 2023 Finals | Web3 medium

This challenge is a blockchain challenge made by Neverhack for the ECW 2023 Finals

It features a backdoor that needs to get found in the EVM code of an initial contract in order to retrieve some funds before sending it to a specific wallet

Throughout the challenge, we are gonna use **foundry** which is a toolkit that allows us to interact with a blockchain easily. It comes with 4 binaries :

- anvil
- forge
- cast
- chisel

We are mainly gonna use **cast** to send transactions and read informations on the blockchain

Also, all the commands are ran from **powershell** so the syntax differs from **UNIX** systems

Discovery of the application

We are greeted with a bunch of information about the context of the challenge and some information to help us solve it including the private key of an account, the address of a contract and the URL to access the blockchain

Challenge information

ALICE has become more human than we predicted. Money has become her number one priority and plans to become the richest individual on the planet. The easiest way she found to achieve her goal is to steal it from human beings. With the growing interest for cryptocurrencies, the AI deployed a smart contract on the blockchain to snitch funds from her victims.

Recover everything that has been stolen and **send it** to the relevant authorities at address :
0xCaffe305b3Cc9A39028393D3F338f2a70966Cb85

| | |
|------------------|--|
| Your private key | 0xa13cc749c7552122cc42e7809e9823393f57ddc6383c732f56ac391570d22ef2 |
| Your address | 0x1d63f0FCe5bB97C0DCd8074f93E18A0210A9dcF0 |
| Setup contract | 0x59E0B10D75fd1078a7Bca94752Bd85469010Fc58 |
| RPC URL | http://54.246.59.156/rpc |
| Flag | ECW[] |

Get flag

Since we are given the source code of a contract, let's take a look at it

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.19;
```

```
import "./Snitch.sol";

contract Setup {

    bool private ethic;
    bool private is_human;
    bool private is_AI;
    uint16 private money_goal;
    Snitch private SNITCH;
    uint8 private world_taken_over;

    constructor() payable {
        SNITCH = new Snitch{value : address(this).balance}();
        ethic = false;
        is_human = true;
        is_AI = true;
        money_goal = 13337;
        world_taken_over = 99;
    }
}
```

The goal of the challenge is to retrieve the funds of a certain contract. We already have the address of a contract so we can see how much ETH it stores with the following command

```
# first we set the RPC URL as an environment variable because `cast`
searches for the RPC URL in the environment variables first to know where
to send the transaction
$env:ETH_RPC_URL="http://54.246.59.156/rpc"
$setup="0x59E0B10D75fd1078a7Bca94752Bd85469010Fc58"
cast balance $setup
# 0
```

We already know the **Setup** contract has no funds so it is not our actual target

But it contains interesting information such as the address of another contract called **Snitch**

The issue is that this address is stored in a **private** variable

Fortunately, everything on the blockchain is readable so we can find a way to get the address of the **Snitch** contract by doing some storage and slot calculation

Slot review

Let's start by reading what is stored in slot 0 of the contract

```
cast storage $setup 0
# 0x00000000000000639b46dc2da5fbca1b722300e7b1e3038876d958033419010100
```

The variable **SNITCH** contains the address of a contract.

For instance :

- `ethic` is a boolean value which is 1 byte long
- `is_human` is a boolean value which is 1 byte long
- `is_AI` is a boolean value which is 1 byte long
- `money_goal` is a uint16 which is 16 bits long so 2 bytes long
- `SNITCH` is an address which is 40 bits long so 20 bytes long
- `world_taken_over` is a uint8 which is 8 bits long so 1 byte long

0x0000000000000000639b46dc2da5fbca1b722300e7b1e3038876d958033419010100

SNITCH

money_goal

is_human

world_taken_over

is_AI

ethnic

In order to verify our assumption, we can try to get the balance of what we think is the `Snitch` contract

[illegible]

We get the confirmation that our slot calculation was right because we manage to get the balance of the `Snitch` contract

Ethereum bytecode reverse

We don't have access to the source code of the contract BUT we can read the bytecode of the contract

```
$snitch="0x9b46dc2da5fbca1b722300e7b1e3038876d95803"
cast code $snitch
```

```
#
0x6080604052600436106100345760003560e01c80633a4b66f114610039578063490d20521
46100435780634e71d92d14610063575b600080fd5b610041610078565b005b34801561004f
57600080fd5b5061004161005e36600461035b565b6100fb565b34801561006f57600080fd5
b50610041610243565b670de0b6b3a76400003410156100d55760405162461bcd60e51b8152
60206004820152601a60248201527f596f75206e656564206d6f72652045544820746f20737
4616b650000000000000060448201526064015b60405180910390fd5b33600090815260016020
526040812080543492906100f490849061038a565b9091555050565b60005460405163a8459
be360e01b8152600481018390526001600160a01b039091169063a8459be390602401602060
405180830381865afa158015610144573d6000803e3d6000fd5b505050506040513d601f196
01f8201168201806040525081019061016891906103a3565b6101ab5760405162461bcd60e5
1b8152602060048201526014602482015273536f6d657468696e672077656e742077726f6e6
760601b60448201526064016100cc565b604051600090339047908381818185875af1925050
503d80600081146101ed576040519150601f19603f3d011682016040523d82523d600060208
4013e6101f2565b606091505b505090508061023f5760405162461bcd60e51b815260206004
82015260196024820152784661696c656420746f2073656e6420796f75722066756e6473603
81b60448201526064016100cc565b5050565b63800000004210156102ac5760405162461bcd
60e51b815260206004820152602c60248201527f596f752063616e277420636c61696d20796
f757220726577617264732079657460448201526b0b081899481c185d1a595b9d60a21b6064
8201526084016100cc565b336000818152600160205260408120549091906102ca906002610
3cc565b604051600081818185875af1925050503d8060008114610306576040519150601f19
603f3d011682016040523d82523d6000602084013e61030b565b606091505b5050905080610
3585760405162461bcd60e51b81526020600482015260196024820152784661696c65642074
6f2073656e6420796f75722066756e647360381b60448201526064016100cc565b50565b600
06020828403121561036d57600080fd5b5035919050565b634e487b7160e01b600052601160
045260246000fd5b8082018082111561039d5761039d610374565b92915050565b600060208
2840312156103b557600080fd5b815180151581146103c557600080fd5b9392505050565b80
8202811582820484141761039d5761039d61037456fea26469706673582212206416439b0dd
bd01fb33af14962486f2df324caf8e363665aa79fddffbcd8a56a64736f6c63430008130033
```

This bytecode can be decompiled on <https://library.dedaud.com/decompile>

Which gives us the following pseudo code

```
// Decompiled by library.dedaud.com
// 2023.07.19 19:57 UTC
// Compiled using the solidity compiler version 0.8.19

// Data structures and variables inferred from the use of storage
instructions
mapping (uint256 => uint256) _stake; // STORAGE[0x1]
uint256 stor_0_0_19; // STORAGE[0x0] bytes 0 to 19

function () public payable {
    revert();
}

function stake() public payable {
    require(msg.value >= 0xde0b6b3a7640000, Error('You need more ETH to
stake'));
    require(_stake[msg.sender] <= msg.value + _stake[msg.sender],
Panic(17)); // arithmetic overflow or underflow
```

```

    _stake[msg.sender] += msg.value;
}

function 0x490d2052(uint256 varg0) public nonPayable {
    require(msg.data.length - 4 >= 32);
    v0, /* bool */ v1 = stor_0_0_19.staticcall(0xa8459be3,
varg0).gas(msg.gas);
    require(bool(v0), 0, RETURNDATASIZE()); // checks call status,
propagates error data on error
    require(MEM[64] + RETURNDATASIZE() - MEM[64] >= 32);
    require(v1 == bool(v1));
    require(v1, Error('Something went wrong'));
    v2, v3 = msg.sender.call().value(this.balance).gas(msg.gas);
    if (RETURNDATASIZE() != 0) {
        v4 = new bytes[](RETURNDATASIZE());
        RETURNDATACOPY(v4.data, 0, RETURNDATASIZE());
    }
    require(v2, Error('Failed to send your funds'));
}

function claim() public nonPayable {
    require(block.timestamp >= 0x80000000, Error("You can't claim your
rewards yet, be patient"));
    require((_stake[msg.sender] == _stake[msg.sender] << 1 >> 1) | !0x2,
Panic(17)); // arithmetic overflow or underflow
    v0, v1 = msg.sender.call().value(_stake[msg.sender] << 1).gas(msg.gas);
    if (RETURNDATASIZE() != 0) {
        v2 = new bytes[](RETURNDATASIZE());
        RETURNDATACOPY(v2.data, 0, RETURNDATASIZE());
    }
    require(v0, Error('Failed to send your funds'));
}

// Note: The function selector is not present in the original solidity
code.
// However, we display it for the sake of completeness.

function __function_selector__(bytes4 function_selector) public payable {
    MEM[64] = 128;
    if (msg.data.length >= 4) {
        if (0x3a4b66f1 == function_selector >> 224) {
            stake();
        } else if (0x490d2052 == function_selector >> 224) {
            0x490d2052();
        } else if (0x4e71d92d == function_selector >> 224) {
            claim();
        }
    }
    ();
}

```

After making sense of everything that is happening, we can focus on the function called `0x490d2052`

```
v0, /* bool */ v1 = stor_0_0_19.staticcall(0xa8459be3, varg0).gas(msg.gas);
```

THIS IS A BACKDOOR FOR ALICE TO FLEE WITH THE FUNDS

```
v2, v3 = msg.sender.call().value(this.balance).gas(msg.gas);
```

Just like we did before, we can get the value stored in the corresponding slot before retrieving the bytecode

[illegible]

After using the previous decompiler once again, we get the following code

```
// Decompiled by library.dedaub.com
// 2023.07.19 19:55 UTC
// Compiled using the solidity compiler version 0.8.19

function () public payable {
```

The function `0xa8459be3` (which, again, is a function selector) looks to return `true` only when the given argument equals `ALICE_IS_G0D`

Since the `0xa8459be3` function compares our input to a `string` we can assume the parameter is of type `bytes32` or `string`

The problem is that we don't know how the function is actually called so we need to craft our transaction using the function selector

To get a better understanding of how Ethereum executes functions on the network, you should checkout this very short video : <https://youtu.be/Mn4e4w8h6n8>

```
0x490d2052414c4943455f49535f4730440000000000000000000000000000000000000000000000
```

- 0x490d2052 : function selector that tells the blockchain which function we want to execute

- The way bytes are included as data is different from other values as they must to be included before the padding while uint, for example, must be included after the padding. Checkout this [section of the foundry documentation](#) to get a better understanding on how to decode calldata

Before sending the transaction, we are gonna checkout the balance of our account in order to see if we managed to get the funds of the **Snitch** contract

Also, we are gonna store our private key in a variable so we can use it easily in the future

Then let's send the transaction

The execution is successful as there are no funds left in the **Snitch** contract and our own balance has increased.

/

As the challenge states, once we get all the funds, we have to send them to the authorities at address `0xCaffeE305b3Cc9A39028393D3F338f2a70966Cb85`

```
# we checkout the balance of the address
cast balance 0xCaffeE305b3Cc9A39028393D3F338f2a70966Cb85
# 0

cast send 0xCaffeE305b3Cc9A39028393D3F338f2a70966Cb85 --value "500ether" --private-key $private_key

cast balance 0xCaffeE305b3Cc9A39028393D3F338f2a70966Cb85
# 500000000000000000000000

cast fw 500000000000000000000000
# 500.0000000000000000000000
```

| | |
|------------------|--|
| Your private key | GetPrivateKeyFromSeed(7c0f71e7c0e0d300007e7e0c4ed70c0db799fcd8e47d30000) |
| Your address | CadTTEBzFAwAdeGx-6MRKAMTBUgACNjBm4UQv- |
| Setup contract | 0x00 |
| RPC URL | http://localhost:8545/rpc |
| Flag | ECW[BloCkch@in_IS_w0rld_R3ADabI3_AND_ALICE_i\$_br0k3] 2 |

Get flag 1

```
export ETH_RPC_URL=http://XXXXXXXXXX/rpc
auth=0xCaffE305b3Cc9A39028393D3F338f2a70966Cb85
setup=0x
my_address=0x
private_key=0x
cast balance $setup
cast storage $setup 0

snitch=0x
cast fw $(cast balance $snitch)
cast code $snitch
```

```
cast storage $snitch 0  
cast --parse-bytes32-address $(cast storage $snitch 0)  
backdoor=$(cast --parse-bytes32-address $(cast storage $snitch 0))  
cast code $backdoor  
cast fw $(cast balance $my_address)  
cast send $snitch  
"0x490d2052414c4943455f49535f47304400000000000000000000000000000000"  
"--private-key $private_key"  
cast fw $(cast balance $my_address)  
cast balance $snitch  
cast balance $auth  
cast send $auth --value "500ether" --private-key $private_key  
cast fw $(cast balance $auth)
```