HACKvent 2019 - Writeup

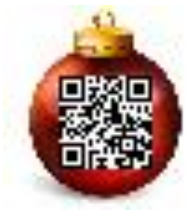
HV19.01 censored

I got this little image, but it looks like the best part got censored on the way. Even the tiny preview icon looks clearer than this! Maybe they missed something that would let you restore the original content?

Solution

Exiftool shows that the image has a large thumbnail. Let's see if this is also censored. We can use mig hty ImageMagick (https://imagemagick.org/) to extract the thumbnail:

```
convert challenge.jpg thumbnail:thumb.jpg
```



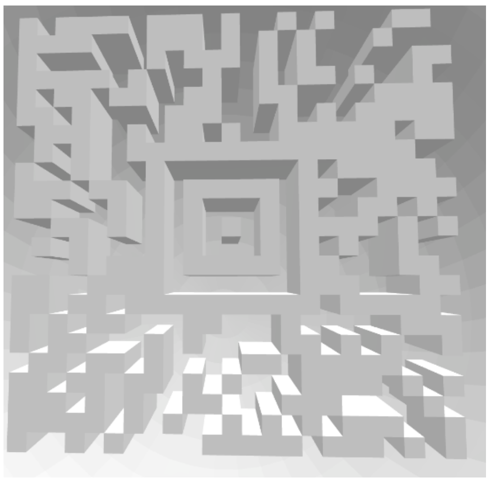
Flag: HV19{just-4-PREview!}

HV19.02 Triangulation

Today we give away decorations for your Christmas tree. But be careful and do not break it.

Solution

The zip contains a 3D plan (.stl file) of a christmas ball. Using any STL viewer (e.g. FreeCAD), we can open the model. Once, we zoom into the inner part of it, we can see a QR code:



Unfortunately, due to the poor contrast and the 3D-effects, my QR code scanner was not able to read the code, so I decided to quickly recreate the pattern in photoshop (I know, manual work sucks, but it works :)).

Flag: HV19{Cr4ck_Th3_B411!}

HV19.03 Hodor, Hodor, Hodor

```
$HODOR: hhodor. Hodor. Hodor!?  = `hodor?!? HODOR!? hodor? Hodor oHodor. hodor? ,
HODOR!?! ohodor!?  dhodor? hodor odhodor? d HodorHodor  Hodor!? HODOR HODOR?
hodor! hodor!? HODOR hodor! hodor? !

hodor?!? Hodor  Hodor Hodor? Hodor  HODOR  rhodor? HODOR Hodor!?  h4Hodor?!?
Hodor?!? 0r hhodor?  Hodor!? oHodor?! hodor? Hodor  Hodor! HODOR Hodor hodor? 64
HODOR Hodor  HODOR!? hodor? Hodor!? Hodor!? .

HODOR?!? hodor- hodorHoOodoOor Hodor?!? OHoOodoOorHooodorrHODOR hodor. oHODOR...
Dhodor- hodor?! HooodorrHODOR HoOodoOorHooodorrHODOR RoHODOR... HODOR!?! 1hodor?!
HODOR... DHODOR- HODOR!?! HooodorrHODOR Hodor- HODORHoOodoOor HODOR!?! HODOR...
DHODORHoOodoOor hodor. Hodor! HoOodoOorHodor HODORHoOodoOor 0Hooodorrhodor
HoOodoOorHooodorrHODOR 0=`;
hodor.hod(hhodor. Hodor. Hodor!? );
```

Solution

This challenge was labelled as fun-programming, so my first guess was that the obscure string could be a program in some sort of esoteric programming language. A quick google search reveals that there is a Hodor programming language. The website links to a GitHub-Repo, which shows that Hodor is just some JavaScript translation. Fortunately, there are already some online Tools, which allow you to run your Hodor code online. I pasted the code and got the following output:

```
Awesome, you decoded Hodors language!

As sis a real h4xx0r he loves base64 as well.

SFYxOXtoMDFkLXRoMy1kMDByLTQyMDQtbGQ0WX0=
```

So we only need to base64-decode the output to get the flag.

Flag: HV19{h01d-th3-d00r-4204-ld4Y}

Bonus

If we look at the code in the Hodor-Repository, we can see that during execution the whole program is translated back to JavaScript and executed using eval. This way, we can easily recover the original JS code:

```
var html = `Awesome, you decoded Hodors language!

As sis a real h4xx0r he loves base64 as well.

SFYxOXtoMDFkLXRoMy1kMDByLTQyMDQtbGQ0WX0=`;
console.log(html);
```

HV19.04 password policy circumvention

Santa released a new password policy (more than 40 characters, upper, lower, digit, special).

The elves can't remember such long passwords, so they found a way to continue to use their old (bad) password:

```
merry christmas geeks
```

Solution

The zip file contains an *.ahk file. A quick google search reveals, that this file-extension is widely used for AutoHotkey, which offers a simple Hotkey-based scripting language for Windows. If we take a look at the code, we see that the phrases of the additional sentence in the challenge description is defined as hotkeys. Therefore, I quickly installed the software, loaded the script and typed these words into a text editor. AutoHotkey applied the logic from the script, and the sentence transformed into the flag (the password that matches the policy).

Flag: HV19{R3memb3r, rem3mber - the 24th 0f December}

HV19.05 Santa Parcel Tracking

To handle the huge load of parcels Santa introduced this year a parcel tracking system. He didn't like the black and white barcode, so he invented a more solemn barcode. Unfortunately the common barcode readers can't read it anymore, it only works with the pimped models santa owns. Can you read the barcode

SP-Tracking: 1337-9999-4555-9

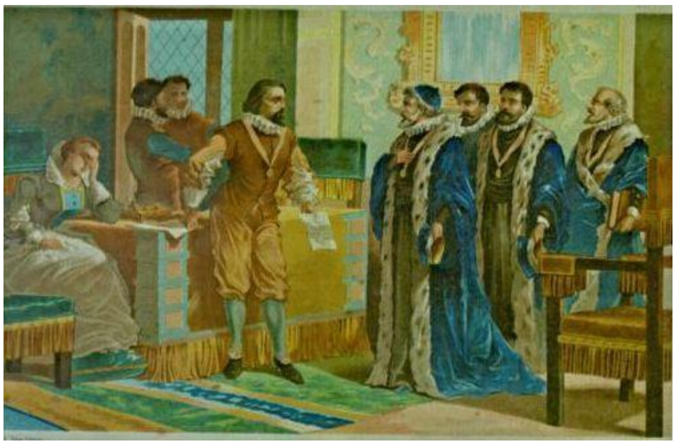
Solution

Scanning the barcode results in "Not the solution", so I assume the flag is stored into the colors. Manual analysis did not help, so I wrote a small C# program, which extracts all the colors from the image, piped them into a file (separated per channel) and took a closer look. As I know that the flag format for HACKvent is HV19{}, I started to look for the related ASCII codes (72, 86, 49, 57). Apparently, this sequence is present in the blue-channel of the barcode (from byte 13 - 48). Finally, I adapted my program to extract the flag:

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;

namespace HV1905
{
    class Program
    {
        static void Main(string[] args)
        {
            var colors = new List<Color>();
            var image = new Bitmap(@"./157de28f-2190-4c6d-a1dc-02ce9e385b5c.png");

            for (var y = 0; y < image.Height; y++)
            {
                for (var x = 0; x < image.Width; x++)
                {
                    var color = image.GetPixel(x, y);
                    if (!colors.Contains(color))
                    {
                        colors.Add(color);
                    }
                }
            }

            var lines = colors.Select(c => {
                return (char)c.B;
            }).ToArray();

            var flag = new string(lines).Substring(13, 35);

            Console.WriteLine(flag);

            Console.ReadLine();
        }

    }
}
```

Flag: HV19{D1fficult_to_g3t_a_SPT_R3ader}

HV19.06 bacon and eggs



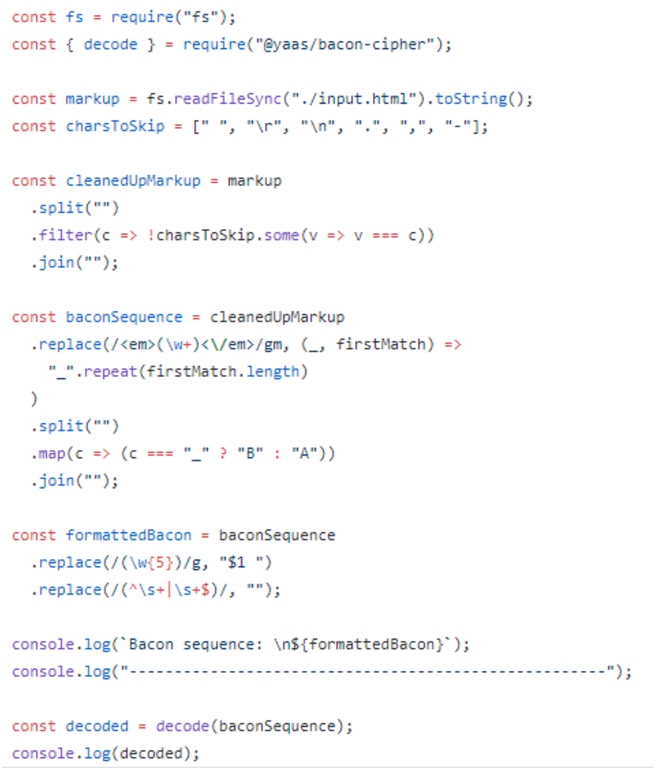
Francis Bacon was an English philosopher and statesman who served as Attorney General and as Lord Chancellor of England. His works are credited with developing the scientific method and remained influential through the scientific revolution. Bacon has been called the father of empiricism. His works argued for the possibility of scientific knowledge based only upon inductive reasoning and careful observation of events in nature. Most importantly, he argued science could be achieved by use of a sceptical and methodical approach whereby scientists aim to avoid misleading themselves. Although his practical ideas about such a method, the Baconian method, did not have a long-lasting influence, the general idea of the importance and possibility of a sceptical methodology makes Bacon the father of the scientific method. This method was a new rhetorical and theoretical framework for science, the practical details of which are still central in debates about science and methodology.

Bacon was the first recipient of the Queen's counsel designation, which was conferred in 1597 when Elizabeth I of England reserved Bacon as her legal advisor. After the accession of James VI and I in 1603, Bacon was knighted. He was later created Baron Verulam in 1618 and Viscount St. Alban in 1621. Because he had no heirs, both titles became extinct upon his death in 1626, at 65 years. Bacon died of pneumonia, with one account by John Aubrey stating that he had contracted the condition while studying the effects of freezing on the preservation of meat. He is buried at St Michael's Church, St Albans, Hertfordshire.

```
Born: January 22
Died: April 9
Mother: Lady Anne
Father: Sir Nicholas
Secrets: unknown
```

Solution

The text is a typeface-based Bacon Cipher. The trick is to convert the individual characters to A's and B's, depending on their typeface. I wrote a small JavaScript program to process the text and decode the Bacon Cipher



```
const fs = require("fs");
const { decode } = require("@yaas/bacon-cipher");

const markup = fs.readFileSync("./input.html").toString();
const charsToSkip = [" ", "\r", "\n", ".", ",", "-"];

const cleanedUpMarkup = markup
  .split("")
  .filter(c => !charsToSkip.some(v => v === c))
  .join("");

const baconSequence = cleanedUpMarkup
  .replace(/<em>(\w+)<\/em>/gm, (_, firstMatch) =>
    "_".repeat(firstMatch.length)
  )
  .split("")
  .map(c => (c === "_" ? "B" : "A"))
  .join("");

const formattedBacon = baconSequence
  .replace(/(\w{5})/g, "$1 ")
  .replace(/(^\s+|\s+$)/, "");

console.log(`Bacon sequence: \n${formattedBacon}`);
console.log("----------------------------------------------------");

const decoded = decode(baconSequence);
console.log(decoded);
```

The output of my program looks like this:

```
Santalikeshisbaconbutalsothisbaconthepasswordishvxbaconcipherissimplebutcoolxrepla
cexwithbracketsanduseuppercaseforallcharacteraaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
a
```

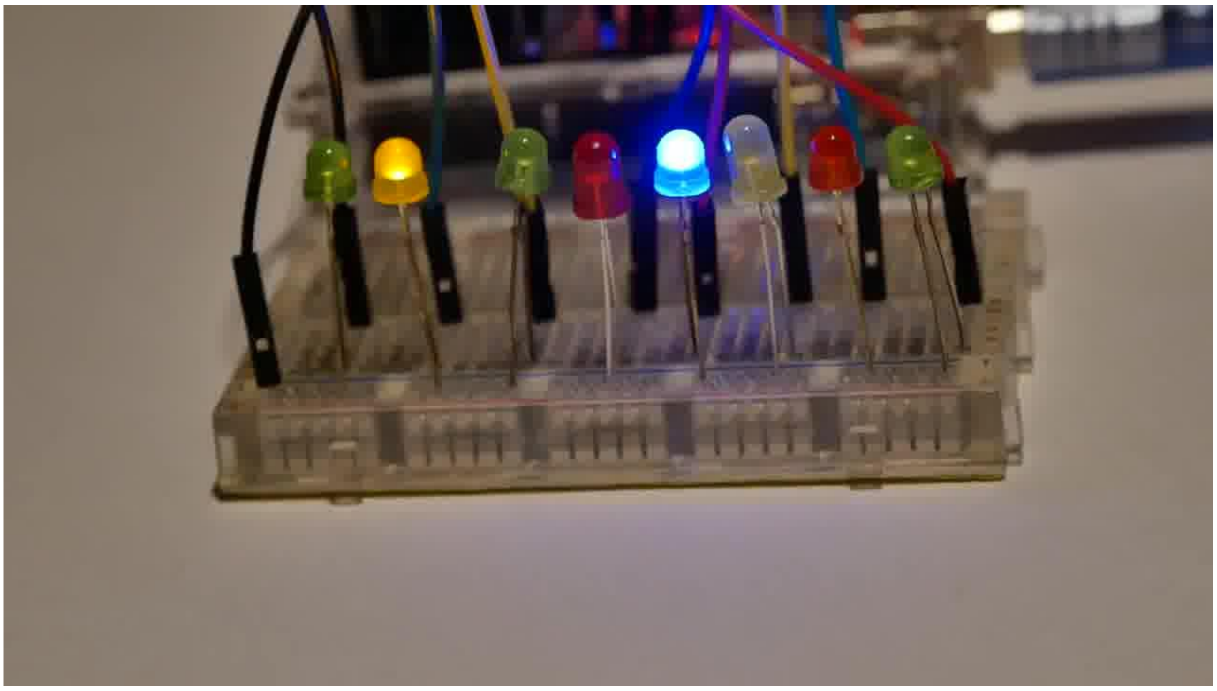
Flag: HV19{BACONCIPHERISSIMPLEBUTCOOL}

HV19.07 Santa Rider

Santa is prototyping a new gadget for his sledge. Unfortunately it still has some glitches, but look for yourself.

Solution

When looking at the middle of the video, we can see that the LEDs start to wildly blink instead of continuing the pattern from the beginning/end. Also, there are exactly 8 LEDs, which could represent eight bits. I stopped the video at the beginning (see photo) and realized that those are indeed the bits of an H.



Using the following ffmpeg command, I extracted all video frames and manually converted the blinking LEDs to bits:

```
ffmpeg -i .\3DULK2N7DcpXFg8qGo9Z9qEQqvaEDpUCBB1v.mp4 image-%d.jpeg
```

After a lot of swearing I ended up with the following bit sequence, which can be converted to the flag:

```
01001000 01010110 00110001 00111001 01111011 00110001 01101101 01011111 01100001
01101100 01110011 00110000 01011111 01110111 00110000 01110010 01101011 00110001
01101110 01100111 01011111 00110000 01101110 01011111 01100001 01011111 01110010
00110011 01101101 00110000 01110100 00110011 01011111 01100011 00110000 01101110
01110100 01110010 00110000 01101100 01111101
```

Flag: HV19{1m_als0_w0rk1ng_0n_a_r3m0t3_c0ntr0l}

HV19.08 SmileNcryptor 4.0

You hacked into the system of very-secure-shopping.com and you found a SQL-Dump with $$-creditcards numbers. As a good hacker you inform the company from which you got the dump. The managers tell you that they don't worry, because the data is encrypted.

Goal

Analyze the "Encryption"-method and try to decrypt the flag.

Solution

When looking at the encrypted credit card numbers, we can see that the characters are close to each other. As also the input characters (0-9) are close, we suspect that this might be a transposition cipher. However, the longer the number gets, the bigger is the offset between the expected source range (0-9) and the target range (Q-b). Therefore, my guess was that the offset increases depending on the position.

I created a small python script to brute force the possible offset values accross all credit card numbers, assuming that there must be one offset that leads to valid credit card numbers. For each position in the encrypted data, I increased the offset by one. Running my program revealed that the only possible offset in this scenario is 30. When using this value as an initial offset, I was able to decrypt the content of the flag-table:

```
#!/usr/bin/python3
import sys

encrypted_flag = r"SlQRUPXWVo\Vuv_n_\ajjce"
encrypted_ccards = [r"QVXSZUVY\ZYYZ[a", r"QOUW[VT^VY]bZ", r"SPPVSSYVV\YY_\\]",
r"RPQRSTUVWXYZ[\]^", r"QTVWRSVUXW[_Z`\b]"]

def decrypt(input, initialShift):
    shift = initialShift
    output = ""

    for c in input:
        target = ord(c) - shift
        try:
            output += chr(target)
        except:
            break

        shift += 1

    return output

for encrypted_card in encrypted_ccards:
    decrypted = decrypt(encrypted_card, 30)
    if len(decrypted) == len(encrypted_card):
        print(decrypted)

print("Flag: HV19{%s}" % (decrypt(encrypted_flag, 30)))
```

Flag: HV19{5M113-420H4-KK3A1-19801}

HV19.09 Santas Quick Response 3.0

Visiting the following railway station has left lasting memories.

Santas brand new gifts distribution system is heavily inspired by it. Here is your personal gift, can you extract the destination path of it?

Solution

Googling the first image directs us to Wolframs Rule 30. This is a special form of a cellular automata. My first guess was that I need to apply this evolution rule on the invalid QR code. I created a small script with Node, but this didn't work.

After wasting a lot of time on this idea, I decided that it's time to try something else. Some parts of the QR code already seem valid, so maybe we only need to change parts of it. When we look at the evolution pattern of rule 30, we see that it forms some sort of pyramid. So why not try to XOR it with our QR code. I updated my node script to calculate the pattern of rule 30 and XORed it with the input image, using various shifts. I also added a QR-code scanning library, to automate the detection step:

```
const CellularAutomata = require("cellular-automata");
const { flatten, cloneDeep } = require("lodash");
const Jimp = require("jimp");
const QrDecoder = require("node-zxing");

async function createImage(filename, data, size) {
  const image = await Jimp.create(size, size);

  for (let y = 0; y < size; y++) {
    for (let x = 0; x < size; x++) {
      const item = data[y * size + x];
      image.setPixelColor(!item ? 0xffffffff : 0xff000000, x, y);
    }
  }

  await image.writeAsync(filename);
}

async function createQrCode(filename, data, size) {
  createImage(filename, data, size);
  const qrDecoder = new QrDecoder();

  return await new Promise(resolve =>
    qrDecoder.decode(filename, (error, code) => resolve([error, code]))
  );
}

function performEvolutionWithRule(data, rule, size, iterations = 1) {
  const cellularAutomata = new CellularAutomata([size]);
  cellularAutomata.setRule(`W${rule}`);
  cellularAutomata.array.data = flatten(data);
  cellularAutomata.iterate(iterations);
  return Array.from(cellularAutomata.array.data);
}
```

```
async function readImageData(filename) {
  const inputImage = await Jimp.read(filename);
  const width = inputImage.getWidth();
  const height = inputImage.getHeight();
  const data = [];
  for (let y = 0; y < height; y++) {
    for (let x = 0; x < width; x++) {
      const color = inputImage.getPixelColor(x, y);
      data.push(color > 255 ? 0 : 1);
    }
  }
  return { width, height, data };
}

function createXorMask(width, height) {
  let row = new Array(width * 2 + 1);
  row.fill(0);
  row[width] = 1;
  const rows = [];
  rows.push(row);

  while (rows.length < height) {
    const transformedRow = performEvolutionWithRule(row, 30, row.length);
    rows.push(transformedRow);
    row = cloneDeep(transformedRow);
  }

  return rows;
}

(async function() {
  const { width, height, data } = await readImageData("./input.png");

  const maskData = createXorMask(width, height);

  let foundFlag = false;
  let startIndex = 0;

  while (!foundFlag && startIndex < maskData[0].length) {
    const flagData = cloneDeep(data);

    for (let y = 0; y < height; y++) {
      for (let x = 0; x < width; x++) {
        flagData[y * width + x] ^= maskData[y][startIndex + x];
      }
    }
    const [_, code] = await createQrCode("./flag.jpg", flagData, width);
    foundFlag = code.length > 0;

    if (foundFlag) {
      console.log(`Got flag: ${code} - Start Index was ${startIndex}`);
    }

    startIndex++;
  }
})();
```

Running the program, results in the following output:

```
Got flag: HV19{Cha0tic_yet-0rdered} - Start Index was 16
```

Flag: HV19{Cha0tic_yet-0rdered}

HV19.10 Guess what

The flag is right, of course

Solution (Version 3)

The challenge contains a 64-Bit ELF file. When running the file, we get prompted for a password. My first guess was to try some static analysis with IDA, but the overall structure of the binary is quite complex. It seems like during runtime, it decrypts some strings in memory and uses the output as part of exec calls. Unfortunately, I had some issues while debugging, so I tried to perform some basic dynamic analysis with ltrace.

```
ltrace -o log.txt ./guess3
```

While examining the output file, I found some very interesting entries:

```
strcpy(0x16291a8, "=")
= 0x16291a8
__ctype_b_loc()
= 0x7f2de02806d0
memset(0x1634e88, '\337', 64)
= 0x1634e88
strcpy(0x1634009, "HV19{Sh3ll_0bfuscat10n_1s_fut1l3"...)
= 0x1634009
strchr("HV19{Sh3ll_0bfuscat10n_1s_fut1l3"..., '$')
= nil
memset(0x1634e88, '\317', 64)
= 0x1634e88
memset(0x1634888, '\337', 36)
= 0x1634888
strcpy(0x1634888, ""HV19{Sh3ll_0bfuscat10n_1s_fut1l"...)
= 0x1634888
__ctype_b_loc()
= 0x7f2de02806d0
strcpy(0x16291c8, "]")
= 0x16291c8
__ctype_b_loc()
```

This already looks like a flag, so why not just append some trailing } and hope this works :)

Flag: HV19{Sh3ll_0bfuscat10n_1s_fut1l3}

HV19.11 Frolicsome Santa Jokes API

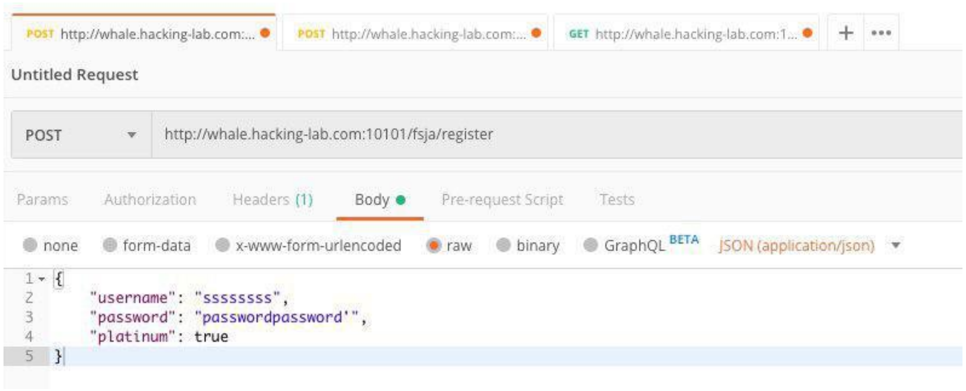
The elves created an API where you get random jokes about santa.

Solution

For this challenge, we get a simple REST API, which allows us to register, login and get random jokes (authenticated users only). The authentication is performed using JSON Web Tokens (JWT). My first guess was to try some common attacks like SQL injection but nothing was successful. I took a closer look at the content of the token, which looks as follows:

```
{
  "user": {
    "username": "testuser",
    "platinum": false
  },
  "exp": 1576086018.984
}
```

The platinum flag looks very suspicious, so why not try to create a new user and try to set this flag to true during registration.

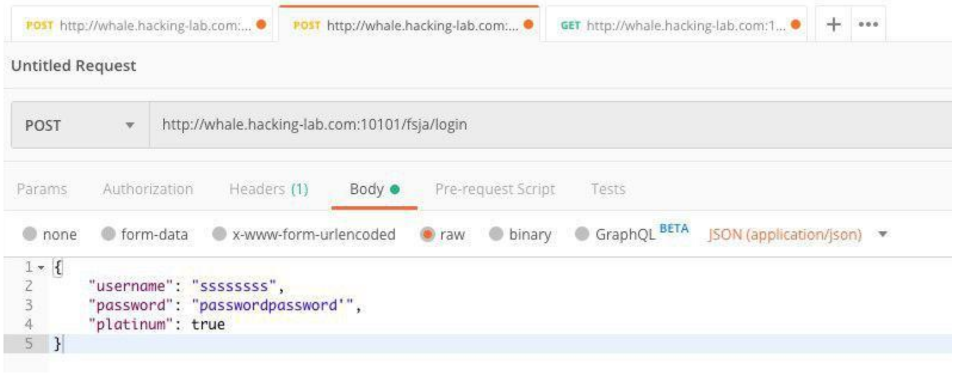


POST http://whale.hacking-lab.com:10101/fsja/register

Body: raw, JSON (application/json)

```
{
    "username": "ssssssss",
    "password": "passwordpassword'",
    "platinum": true
}
```

Now let's login:

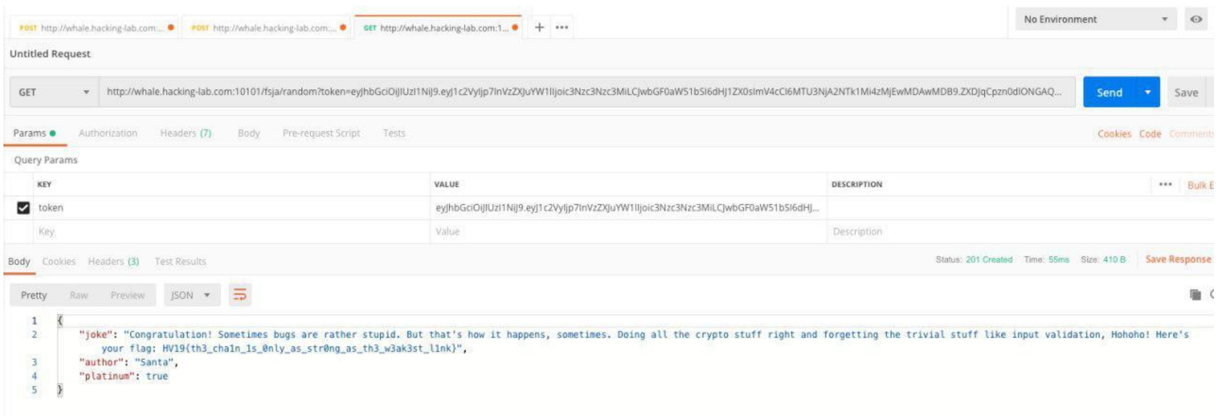


POST http://whale.hacking-lab.com:10101/fsja/login

Body: raw, JSON (application/json)

```
{
    "username": "ssssssss",
    "password": "passwordpassword'",
    "platinum": true
}
```

And request a random token:



GET http://whale.hacking-lab.com:10101/fsja/random?token=eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjp7InVzZXJuYW1lIjoic3Nzc3Nzc3MiLCJwbGF0aW51bSI6dHJ1ZX0sImV4cCI6MTU3NjA2NTk1Mi4zMjEwMDAwMDB9.ZXDJqCpzn0dIONGAQ...

Query Params: token = eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjp7InVzZXJuYW1lIjoic3Nzc3Nzc3MiLCJwbGF0aW51bSI6dHJ...

Status: 201 Created Time: 55ms Size: 410 B

```
{
    "joke": "Congratulation! Sometimes bugs are rather stupid. But that's how it happens, sometimes. Doing all the crypto stuff right and forgetting the trivial stuff like input validation, Hohoho! Here's your flag: HV19{th3_cha1n_1s_0nly_as_str0ng_as_th3_w3ak3st_l1nk}",
    "author": "Santa",
    "platinum": true
}
```

Flag: HV19{th3_cha1n_1s_0nly_as_str0ng_as_th3_w3ak3st_l1nk}

HV19.12 back to basic

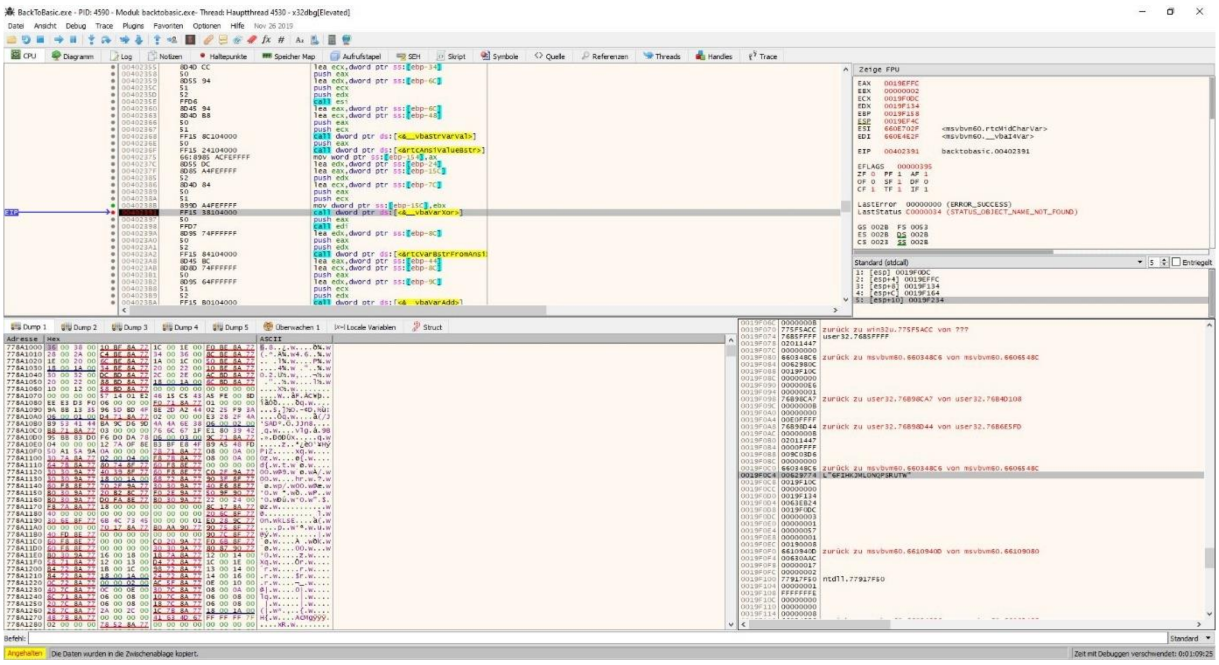
Santa used his time machine to get a present from the past. get your rusty tools out of your cellar and solve this one!

Solution

For this challenge, we get an old school Visual Basic binary. My first thought was that this should be easy, because VB is similar to C#, which can be simply decompiled using public tools. However, it turns out that this is not as easy for old VB binaries (although they work quite similar). After trying a couple of (freemium) VB decompilers, I decided to switch back to good old IDA.

After a while of reversing, I found the function which reacts to changes of the text in the input field (starting at 0x401F80). It then performs a 3-step check, to validate the flag. The first check is to verify the flag format, checking for it to start with HV19. Then it also compares the length of the input, which needs to be 33. Finally, it loops through the characters of the flag content (starting from index 6) and xors them. The result of this operation, gets compared to the string 6klzic<=bPBtdvff'y�FI~on//N. If the values match, the input was correct.

However, using only static analysis I was not really able to grasp how this loop calculates the encrypted flag, so I decided to debug the program using x32dbg. By setting a breakpoint at the xor instruction (at 0x402391) and examining the values on the stack, I found out that the encryption simply xors each character with its index:



```
BackToBasic.exe - PID: 4590 - Modul: backtobasic.exe- Thread: Hauptthread 4530 - x32dbg[Elevated]

00402355   8D4D CC          lea ecx,dword ptr ss:[ebp-34]
00402358   50               push eax
00402359   8D55 94          lea edx,dword ptr ss:[ebp-6C]
0040235C   51               push ecx
0040235D   52               push edx
0040235E   FFD6             call esi
00402360   8D45 94          lea eax,dword ptr ss:[ebp-6C]
00402363   8D4D B8          lea ecx,dword ptr ss:[ebp-48]
00402366   50               push eax
00402367   51               push ecx
00402368   FF15 8C104000    call dword ptr ds:[<&__vbaStrVarVal>]
0040236E   50               push eax
0040236F   FF15 24104000    call dword ptr ds:[<&rtcAnsiValueBstr>]
00402375   66:8985 ACFEFFFF mov word ptr ss:[ebp-154],ax
0040237C   8D55 DC          lea edx,dword ptr ss:[ebp-24]
0040237F   8D85 A4FEFFFF    lea eax,dword ptr ss:[ebp-15C]
00402385   52               push edx
00402386   8D4D 84          lea ecx,dword ptr ss:[ebp-7C]
00402389   50               push eax
0040238A   51               push ecx
0040238B   899D A4FEFFFF    mov dword ptr ss:[ebp-15C],ebx
00402391   FF15 38104000    call dword ptr ds:[<&__vbaVarXor>]
00402397   50               push eax
00402398   FFD7             call edi
0040239A   8D95 74FFFFFF    lea edx,dword ptr ss:[ebp-8C]
004023A0   50               push eax
004023A1   52               push edx
004023A2   FF15 84104000    call dword ptr ds:[<&rtcVarBstrFromAnsi>]
004023A8   8D45 BC          lea eax,dword ptr ss:[ebp-44]
004023AB   8D8D 74FFFFFF    lea ecx,dword ptr ss:[ebp-8C]
004023B1   50               push eax
004023B2   8D95 64FFFFFF    lea edx,dword ptr ss:[ebp-9C]
004023B8   51               push ecx
004023B9   52               push edx
004023BA   FF15 B0104000    call dword ptr ds:[<&__vbaVarAdd>]
```

```
Zeige FPU
EAX   0019EFFC
EBX   00000002
ECX   0019F0DC
EDX   0019F134
EBP   0019F158
ESP   0019EF4C
ESI   660E702F     <msvbvm60.rtcMidCharVar>
EDI   660E4E2F     <msvbvm60.__vbaI4Var>

EIP   00402391     backtobasic.00402391

EFLAGS   00000395
ZF 0  PF 1  AF 1
OF 0  SF 1  DF 0
CF 1  TF 1  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus C0000034 (STATUS_OBJECT_NAME_NOT_FOUND)

GS 002B  FS 0053
ES 002B  DS 002B
CS 0023  SS 002B

Standard (stdcall)   5   Entriegelt
1: [esp] 0019F0DC
2: [esp+4] 0019EFFC
3: [esp+8] 0019F134
4: [esp+C] 0019F164
5: [esp+10] 0019F234
```

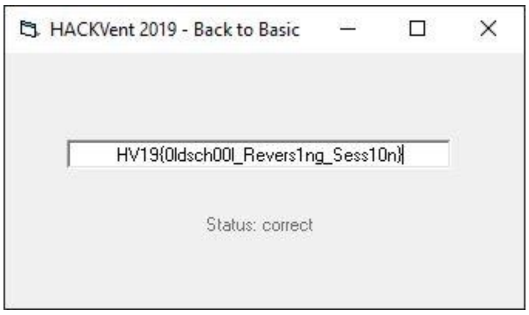
```
Adresse  Hex                                              ASCII
778A1000 36 00 38 00 10 BF 8A 77 1C 00 1E 00 F0 BE 8A 77  6.8..¿.w....ð¾.w
778A1010 28 00 2A 00 C4 BE 8A 77 34 00 36 00 8C BE 8A 77  (.*.Ä¾.w4.6..¾.w
778A1020 1E 00 20 00 6C BE 8A 77 1A 00 1C 00 50 BE 8A 77  .. .l¾.w....P¾.w
778A1030 18 00 1A 00 34 BE 8A 77 20 00 22 00 10 BE 8A 77  ....4¾.w .".. ¾.w
778A1040 30 00 32 00 DC BD 8A 77 2C 00 2E 00 AC BD 8A 77  0.2.Ü½.w,...¬½.w
778A1050 20 00 22 00 88 BD 8A 77 18 00 1A 00 6C BD 8A 77   ."..½.w....l½.w
778A1060 10 00 12 00 58 BD 8A 77 00 00 00 00 00 00 00 00  ....X½.w........
778A1070 00 00 00 00 57 14 01 E2 46 15 C5 43 A5 FE 00 8D  ....W..âF.ÅC¥þ..
778A1080 EE E3 D3 F0 06 00 00 00 F0 71 8A 77 01 00 00 00  îãÓð....ðq.w....
778A1090 9A 8B 13 35 96 5D BD 4F 8E 2D A2 44 02 25 F9 3A  ...5.]½O.-¢D.%ù:
778A10A0 06 00 01 00 D4 71 8A 77 02 00 00 00 E3 28 2F 4A  ....Ôq.w....ã(/J
778A10B0 B9 53 41 44 BA 9C D6 9D 4A 4A 6E 38 06 00 02 00  ¹SAD°.Ö.JJn8....
778A10C0 88 71 8A 77 03 00 00 00 76 6C 67 1F E1 80 39 42  .q.w....vlg.á.9B
778A10D0 95 BB 83 D0 F6 D0 DA 78 06 00 03 00 9C 71 8A 77  .».ÐöÐÚx.....q.w
778A10E0 04 00 00 00 12 7A 0F 8E B3 BF E8 4F B9 A5 48 FD  .....z..³¿èO¹¥Hý
778A10F0 50 A1 5A 9A 0A 00 00 00 28 71 8A 77 08 00 0A 00  P¡Z.....(q.w....
778A1100 30 7A 8A 77 02 00 04 00 F8 7B 8A 77 08 00 0A 00  0z.w....ø{.w....
778A1110 64 7B 8A 77 80 74 8F 77 60 F8 8E 77 00 00 00 00  d{.w.t.w`ø.w....
778A1120 30 30 9A 77 40 39 8F 77 60 F8 8E 77 C0 2F 9A 77  00.w@9.w`ø.wÀ/.w
778A1130 30 30 9A 77 18 00 1A 00 68 72 8A 77 90 3F 8F 77  00.w....hr.w.?.w
778A1140 60 F8 8E 77 70 2F 9A 77 30 30 9A 77 40 E6 8E 77  `ø.wp/.w00.w@æ.w
778A1150 80 30 9A 77 20 B2 8C 77 F0 2E 9A 77 50 9F 90 77  .0.w ².wð..wP..w
778A1160 80 30 9A 77 D0 FA 8E 77 B0 30 9A 77 22 00 24 00  .0.wÐú.w°0.w".$.
778A1170 F8 7A 8A 77 18 00 00 00 00 00 00 00 8C 17 8A 77  øz.w...........w
778A1180 40 00 00 00 00 00 00 00 00 00 00 00 20 6C 8F 77  @........... l.w
778A1190 30 6E 8F 77 6B 4C 73 45 00 00 00 01 E0 28 9C 77  0n.wkLsE....à(.w
778A11A0 00 00 00 00 70 17 8A 77 B0 AA 90 77 90 75 8F 77  ....p..w°ª.w.u.w
778A11B0 40 FD 8E 77 00 00 00 00 00 00 00 00 90 7C 8F 77  @ý.w.........|.w
778A11C0 60 F8 8E 77 00 00 00 00 C0 20 9A 77 F0 6B 8F 77  `ø.w....À .wðk.w
778A11D0 60 F8 8E 77 00 00 00 00 30 30 9A 77 80 87 90 77  `ø.w....00.w...w
778A11E0 80 30 9A 77 16 00 18 00 18 7A 8A 77 12 00 14 00  .0.w.....z.w....
778A11F0 58 71 8A 77 12 00 13 00 D4 72 8A 77 1C 00 1E 00  Xq.w....Ôr.w....
778A1200 84 72 8A 77 18 00 1C 00 98 72 8A 77 13 00 14 00  .r.w.....r.w....
778A1210 84 72 8A 77 18 00 1A 00 24 72 8A 77 14 00 16 00  .r.w....$r.w....
778A1220 0C 72 8A 77 00 00 02 00 AC 5F 8A 77 0E 00 10 00  .r.w....¬_.w....
778A1230 40 7C 8A 77 0C 00 0E 00 30 7C 8A 77 08 00 0A 00  @|.w....0|.w....
778A1240 6C 71 8A 77 06 00 08 00 10 7C 8A 77 06 00 08 00  lq.w.....|.w....
778A1250 20 7C 8A 77 06 00 08 00 18 7C 8A 77 06 00 08 00   |.w.....|.w....
778A1260 28 7C 8A 77 2A 00 2C 00 1C 7B 8A 77 18 00 1A 00  (|.w*.,..{.w....
778A1270 48 7B 8A 77 00 00 00 00 41 63 4D 67 FF FF FF 7F  H{.w....AcMgÿÿÿ.
778A1280 02 00 00 00 78 52 8A 77 00 00 00 00 00 00 00 00  ....xR.w........
```

```
0019F06C 0000000B
0019F070 775F5ACC zurück zu win32u.775F5ACC von ???
0019F074 7685FFFF user32.7685FFFF
0019F078 02011447
0019F07C 00000000
0019F080 660348C6 zurück zu msvbvm60.660348C6 von msvbvm60.6606548C
0019F084 0062980C
0019F088 0019F10C
0019F08C 00000000
0019F090 000000E6
0019F094 00000001
0019F098 76B98CA7 zurück zu user32.76B98CA7 von user32.76B4D108
0019F09C 0000000B
0019F0A0 00000000
0019F0A4 00E0FFFF
0019F0A8 76B98D44 zurück zu user32.76B98D44 von user32.76B6E5FD
0019F0AC 0000000B
0019F0B0 02011447
0019F0B4 0000FFFF
0019F0B8 009C03D6
0019F0BC 00000000
0019F0C0 660348C6 zurück zu msvbvm60.660348C6 von msvbvm60.6606548C
0019F0C4 00629774 L"6FIHKJMLONQPSRUTW"
0019F0C8 0019F10C
0019F0CC 00000000
0019F0D0 0019F134
0019F0D4 0063EB24
0019F0D8 0019F0DC
0019F0DC 00000003
0019F0E0 00000001
0019F0E4 00000057
0019F0E8 00000001
0019F0EC 00190008
0019F0F0 6610940D zurück zu msvbvm60.6610940D von msvbvm60.66109080
0019F0F4 00630AAC
0019F0F8 00000017
0019F0FC 00000002
0019F100 77917F50 ntdll.77917F50
0019F104 00000001
0019F108 FFFFFFFE
0019F10C 00000000
0019F110 00000000
0019F114 00000008
```

Angehalten   Die Daten wurden in die Zwischenablage kopiert.   Zeit mit Debuggen verschwendet: 0:01:09:25

Using this information, I wrote a small python script, which calculates the flag:

```python
#!/usr/bin/python3
encrypted_flag = "6klzic<=bPBtdvff'y FI~on//N"
decrypted_flag = "HV19{"
for i in range(len(encrypted_flag)):
    char = chr(ord(encrypted_flag[i]) ^ (6 + i))
    decrypted_flag += char

decrypted_flag += '}'
print(decrypted_flag)
```



HACKVent 2019 - Back to Basic

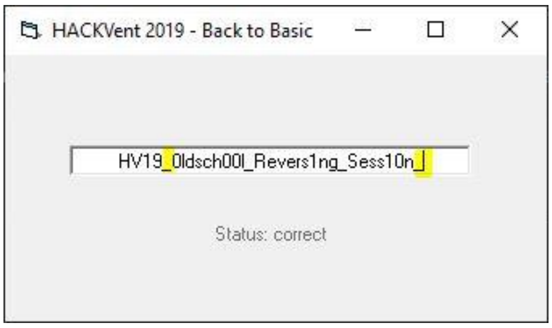
HV19{0ldsch00l_Revers1ng_Sess10n}

Status: correct

Flag: HV19{0ldsch00l_Revers1ng_Sess10n}

Fun Fact

The validation step completely ignores the curly braces around the flag content, so the program also accepts any other character in these positions:



HV19_0ldsch00l_Revers1ng_Sess10n_

Status: correct

HV19.13 TrieMe

Switzerland's national security is at risk. As you try to infiltrate a secret spy facility to save the nation you stumble upon an interesting looking login portal.

Can you break it and retrieve the critical information?

Solution

For this challenge, we get access to a webpage with a single input field, and some Java Code. The code represents a session scoped Java Bean, which probably handles all the incoming requests. We see that for each client, the app creates a new session with a PatriciaTrie. This special sort of tree stores the data in inner leaves, and not only in leaves. Therefore, it requires less memory to store the same data. Morevover, the app automatically stores a security token (auth_token_4835989) in the trie.

When submitting a request, we assume that our input gets passed to the addTrie method, which adds it to the trie in the following way:

```java
public void setTrie(String note) {
    trie.put(unescapeJava(note), 0);
}
```

Then, the application checks if the current session has admin permission, which means that the trie does not contain a security token. If that is the case, it returns the flag. However, at a first glance this seems to be almost impossible, because whenever we make a new request, the security token gets automatically inserted into the trie. Our only way to interact with the trie is by inserting new items. After digging through the Wikipedia pages and the Java-Documentation for quite a while, I learned that it could be possibly to trick the containsKey check by inserting items which are almost equal to the security token, but get matched before. Also, the unescapeJava call in the setTrie method looks very suspicious. Probably, we need to add some control characters to perform this task.

To simplifiy my testing procedure, I created a small Java-Program which mimics the server-side behavior. After some tries, I discovered that appending a null-byte to the auth token does the trick: **auth_token_4835989\0**

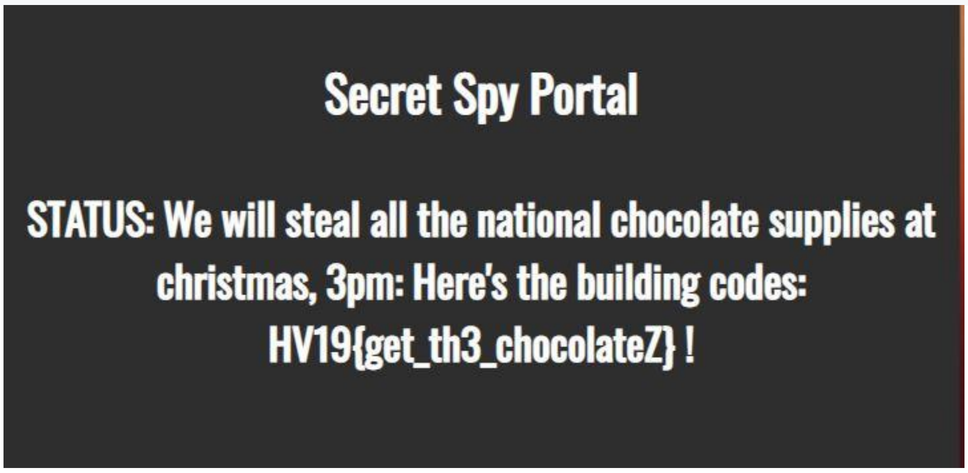
The final program to verify this behavior looks as follows:

```java
package at.hackvent19;
import static org.apache.commons.lang3.StringEscapeUtils.unescapeJava;
import org.apache.commons.collections4.trie.PatriciaTrie;

public class Main {
        private static final String securitytoken = "auth_token_4835989";
```

```
          @SuppressWarnings("deprecation")
          public static void main(String[] args) {
                    PatriciaTrie<Integer> trie = new PatriciaTrie<Integer>();
                    trie.put(securitytoken, 0);
                    String input = "auth_token_483598\u0039";
                    input = "auth_token_4835989\0";

                    trie.put(unescapeJava(input), 0);

                    System.out.println(unescapeJava(input));
                    System.out.println(unescapeJava(input).equals(securitytoken));

                    System.out.println("Contains input: " + trie.containsKey(input));
                    System.out.println("Contains escaped input: " +
trie.containsKey(unescapeJava(input)));
                    System.out.println("Contains token: " + trie.containsKey(securitytoken));
          }
}
```

The output looks as follows:

```
auth_token_4835989
false
Contains input: true
Contains escaped input: true
Contains token: false
```


Secret Spy Portal

STATUS: We will steal all the national chocolate supplies at christmas, 3pm: Here's the building codes: HV19{get_th3_chocolateZ} !

Flag: HV19{get_th3_chocolateZ}

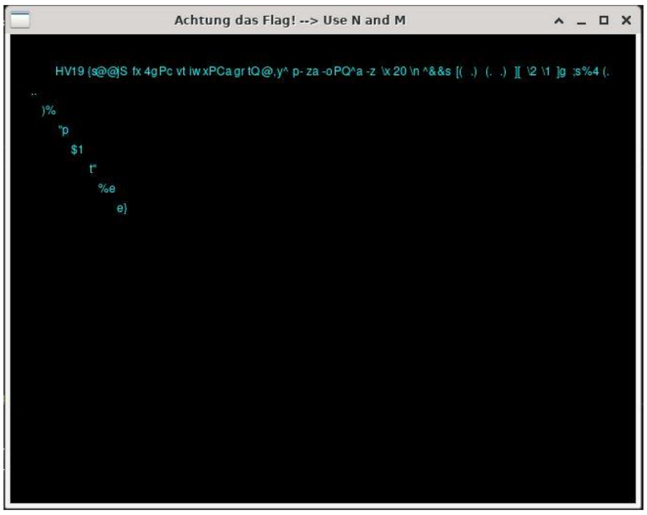
HV19.14 Achtung das Flag

Let's play another little game this year. Once again, I promise it is hardly obfuscated.

Solution

For this challenge, we get a perl implementation of the game Achtung die Kurve. Instead of opponents, we need to collect parts of the flag (similar to snake). Of course, solving the challenge by simply playing the game is really hard (if not impossible) and also boring. Therefore, I decided to patch the script. In the first step, I started to format the script and rename variables, to get some general overview what is going on.

Using this approach, it was quite easy to figure out which conditions force the game to prematurely exit (because they lead to a call of the cancel function). This way, I was able to prevent the game from stopping once I hit a wall or some part of my snake. Moreover, I made sure the game still renders already collected flag fragments (by commenting out the call to delete) and adapted the placement of the characters. Finally, I patched updated the condition to always render the next part of the flag, no matter if the previous one was already collected. After all these changes, the game is essentially self-solving and prints the flag without further human interaction:



```
HV19{s@@jS fx 4gPc vt iwxPCagr tQ@,y^ p- za -oPQ^a -z \x 20 \n ^&&s [(  .)  (.  .) ][ \2 \1 ]g ;s%4 (.
..
)%
"p
$1
t"
%e
e}
```

The final patched version of my program looks as follows:

```
use Tk;
use MIME::Base64;
chomp(($a,$a,$b,$c,$f,$u,$z,$y,$r,$r,$u)=<DATA>);
sub M{
    $M=shift;##
    @m=keys %::;
    (grep{(unpack("%32W*",$_).length($_))eq$M}@m)[0]
};
$zvYPxUpXMSsw=0x1337C0DE;###
/_help_me_/;
$PMMtQJOcHm8eFQfdsdNAS20=sub{
    $zvYPxUpXMSsw=($zvYPxUpXMSsw*16807)&0xFFFFFFFF;
};

($a1Ivn0ECw49I5I0oE0='07&3-"11*/(')=~y$!-=$`-~$;
($Sk61A7pO='K&:P3&44')=~y$!-=$`-~$;
m/Mm/g;
($sk6i47pO='K&:R&-&"4&')=~y$!-=$`-~$;;;;
$d28Vt03MEbdY0=sub{
    pack('n',$fff[$S9cXJIGB0BWce++] ^($PMMtQJOcHm8eFQfdsdNAS20->()&0xDEAD));
};
'42';
($vgOjwRk4wIo7_=MainWindow->new)->title($r);

($vMnyQdAkfgIIik=$vgOjwRk4wIo7_->Canvas("-$a"=>640,"-$b"=>480,"-$u"=>$f))->pack;
@p=(42,42);
$cqI=$vMnyQdAkfgIIik->createLine(@p,@p,"-$y"=>$c,"-$a"=>3);;;
$S9cXJIGB0BWce=0;
$_2kY10=0;
$_8NZQooI5K4b=0;
$Sk61A7p0=0;
$MMM__;
$_=M(120812).'/'.M(191323).M(133418).M(98813).M(121913)
.M(134214).M(101213).'/'.M(97312).M(6328).M(2853).'+'.M(4386);
s|_||gi;
@fff=map{unpack('n',$::{M(122413)}->($_))}m:...:g;
$posx = 20;
$posy = 20;
($T=sub{
    #$vMnyQdAkfgIIik->delete($t);
    $pos_t = $PMMtQJOcHm8eFQfdsdNAS20->();$posx += 15;
    $pos_y = $PMMtQJOcHm8eFQfdsdNAS20->();

    $newPosX = $posx%600+20;
    $idealPosX = $posx+20;

    if ($newPosX < $idealPosX) {$posy += 20;}
    $t=$vMnyQdAkfgIIik->#FOO
    createText($posx%600+20,$posy%440+20,#Perl!!
    "-text"=>$d28Vt03MEbdY0->(),"-$y"=>$z);
})->();
$HACK;
$i=$vMnyQdAkfgIIik->repeat(25, sub{
    $_=($_8NZQooI5K4b+=0.1*$Sk6lA7p0);;
    $p[0]+=3.0*cos;
    $p[1]-=3*sin;;
    #($p[0]>1 && $p[1]>1 && $p[0]<639 && $p[1]<479) || $i->cancel();
    00;
    $q=($vMnyQdAkfgIIik->find($a1Ivn0ECw49I5I0oE0,$p[0]-1,$p[1]-1, $p[0]+1,$p[1]+1)||[])->[0];
    #$q==$t&&$T->();
    $T->();
    #$vMnyQdAkfgIIik->insert($cqI,'end',\@p);
    ($S9cXJIGB0BWce > 44) && $i->cancel();
   # ($q == $cqI || $S9cXJIGB0BWce > 44) && $i->cancel();
});

$KE=5;
$vgOjwRk4wIo7_->bind("<$Sk61A7pO-n>"=>sub{
    $Sk6lA7p0=1;
});
$vgOjwRk4wIo7_->bind("<$Sk61A7pO-m>"=>sub{
    $Sk6lA7p0=-1;
});
$vgOjwRk4wIo7_#%"
    ->bind("<$sk6i47pO-n>"=>sub{
        $Sk6lA7p0=0 if$Sk6lA7p0>0;
    });
$vgOjwRk4wIo7_->bind("<$sk6i47pO"
    ."-m>"=>sub{
        $Sk6lA7p0=0 if $Sk6lA7p0<0;
    });
$::{M(7998)}->();
$M_decrypt=sub{'HACKVENT2019'};
__DATA__
The cake is a lie!
width
height
orange
black
green
cyan
fill
Only perl can parse Perl!
Achtung das Flag! --> Use N and M
background
M'); DROP TABLE flags; --
Run me in Perl!
__DATA__
```

Flag: HV19{s@@jSfx4gPcvtiwxPCagrtQ@,y^p-za-oPQ^a-z\x20\n^&&s[(.)(..)][\2\1]g;s%4(...)%"p$1t"%ee}

HV19.15 Santa's Workshop

The Elves are working very hard. Look at http://whale.hacking-lab.com:2080/ to see how busy they are.

Solution

The linked website contains a gift counter, which automatically increases. When taking a look at the source code of the site, we can see that it opens a MQTT connection via Websockets to retrieve this information. The code, which initiates the connection looks very suspicious:

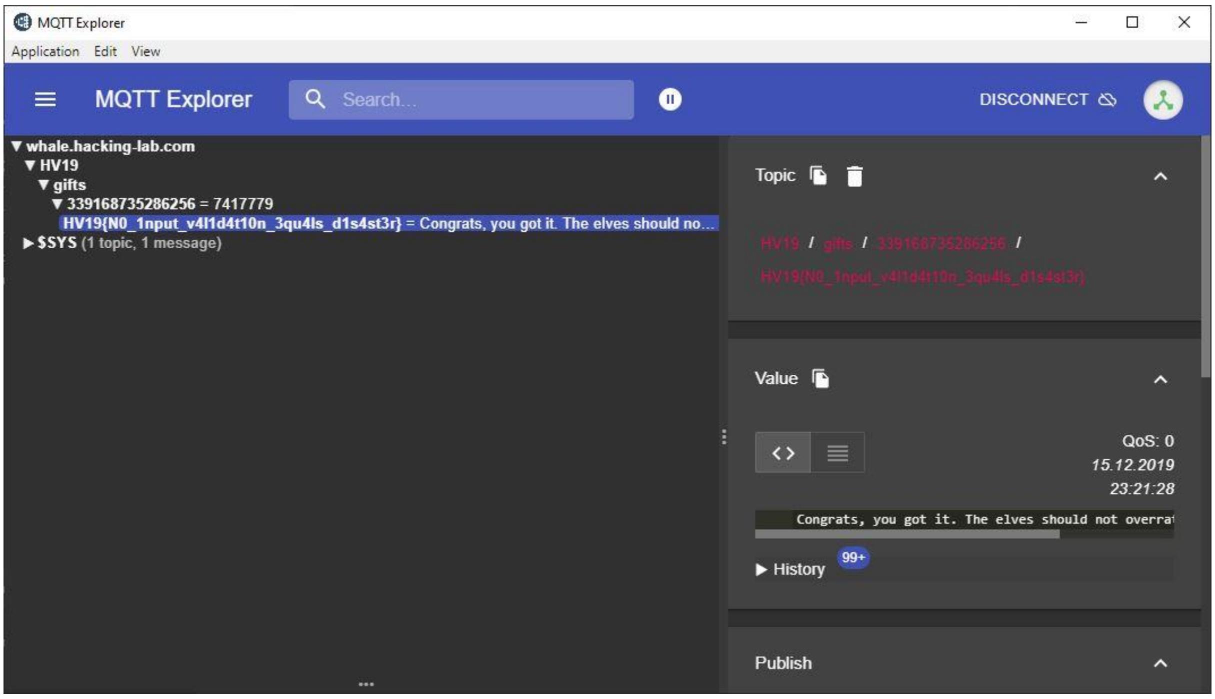
```
var mqtt;
var reconnectTimeout = 100;
var host = "whale.hacking-lab.com";
var port = 9001;
var useTLS = false;
var username = "workshop";
var password = "2fXc7AWINBXyruvKLiX";
var clientid = localStorage.getItem("clientid");
if (clientid == null) {
  clientid = ("" + Math.round(Math.random() * 10000000000000000)).padStart(
    16,
    "0"
  );
  localStorage.setItem("clientid", clientid);
}
var topic = "HV19/gifts/" + clientid;
// var topic = 'HV19/gifts/'+clientid+'/flag-tbd';
var cleansession = true;
```

From this, we see that the flag is stored in one of the topics. Using MQTT-Explorer, I was able to manually connect to the service (the client-id can be easily retrieved by looking at the local storage). Of course, it is not possible to simply query the flag topic, but the $SYS entry contains an interesting message:

```
mosquitto version 1.4.11 (We elves are super-smart and know about CVE-2017-7650
and the POC. So we made a genious fix you never will be able to pass. Hohoho)
```

So this challenge seems to be related to CVE-2017-7650. Probably, the elves made a faulty patch. When looking at the CVE description, we see that for the affected mosquitto versions, it was possible to bypass pattern based ACLs, by using a (# or +) sign as client id. Those signs are used as wildcards. The official patch rejects all client-ids which contain this character. Some testing shows, that our server only rejects client-ids which directly start with a wildcard character. Maybe, we can use them in some other place to retrieve the flag topic.

After a number of failed attempts, I tried to use a client id of 339168735286256/# (essentially appending /# to an ordinary client id) succeeded. The wildcard matches all sub-topics for this client, so I was able to access the flag.



```
▼ whale.hacking-lab.com
  ▼ HV19
    ▼ gifts
      ▼ 339168735286256 = 7417779
        HV19{N0_1nput_v4l1d4t10n_3qu4ls_d1s4st3r} = Congrats, you got it. The elves should no...
  ▶ $SYS (1 topic, 1 message)
```

HV19 / gifts / 339168735286256 / HV19{N0_1nput_v4l1d4t10n_3qu4ls_d1s4st3r}

15.12.2019 23:21:28

```
Congrats, you got it. The elves should not overrat
```

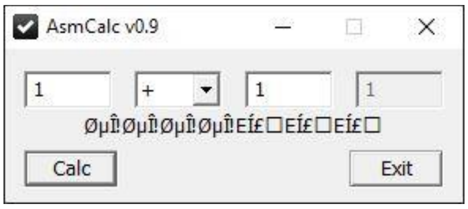
Flag: HV19{N0_1nput_v4l1d4t10n_3qu4ls_d1s4st3r}

HV19.16 B0rked Calculator

Santa has coded a simple project for you, but sadly he removed all the operations. But when you restore them it will print the flag!

Solution

For this challenge, we get a PE32 executable. The exe is a simple calculator program, which allows us to do the 4 basic arithmetic operations (add, subtract, multiply, divide). However, none of the operations works as expected. Moreover, we see that the calculator outputs some weird characters after we do a simple calculation:

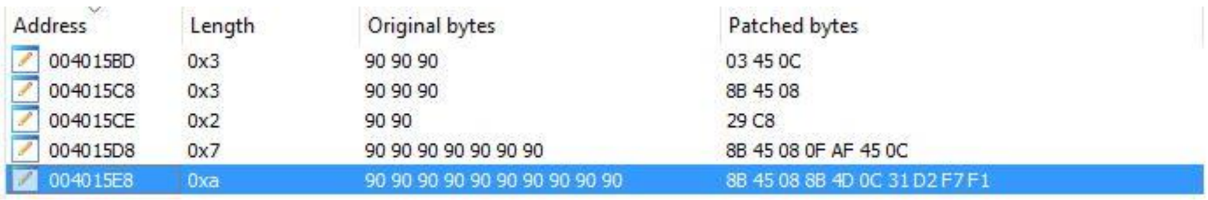


```
ØµÎØµÎØµÎØµÎEí£□Eí£□Eí£□
```

As the challenge description already mentions, the operations have been removed. Probably, if we successfully readd them, this line will contain the flag. I opened the application in IDA and was quickly able to find the stub-functions for the required operations:

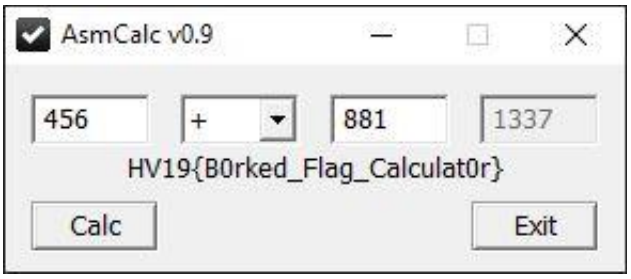
- add: 0x4015B6
- subtract: 0x4015C4
- multiply: 0x4015D4
- divide: 0x4015E4

However, the content of this functions was completely nop-ed out. Time to dig out some basic assembly knowledge and restore the correct instructions! By applying the following patches, I was able to recover the functionality of the calculator:



Address	Length	Original bytes	Patched bytes
004015BD	0x3	90 90 90	03 45 0C
004015C8	0x3	90 90 90	8B 45 08
004015CE	0x2	90 90	29 C8
004015D8	0x7	90 90 90 90 90 90 90	8B 45 08 0F AF 45 0C
004015E8	0xa	90 90 90 90 90 90 90 90 90 90	8B 45 08 8B 4D 0C 31 D2 F7 F1

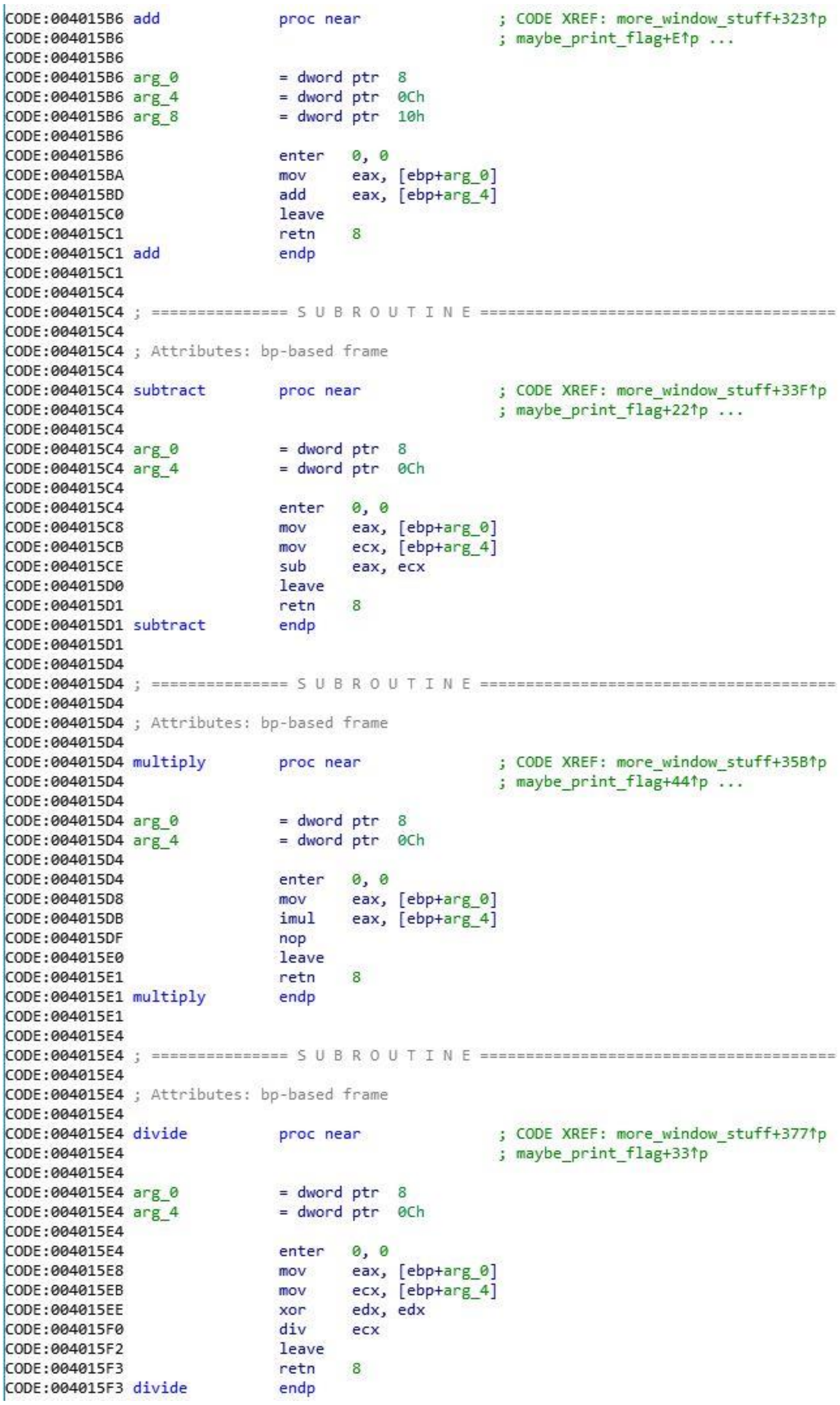
When looking at the xrefs of these functions, we see that they are not only used to calculate the equations entered by the user, but also in some other function (0x401519). This function is used when calculating the flag. If we start the patched executable and do some simple calculation, we are able to get the flag:



```
HV19{B0rked_Flag_Calculat0r}
```

Flag: HV19{B0rked_Flag_Calculat0r}

The final assembly looks as follows:



```
CODE:004015B6 add             proc near               ; CODE XREF: more_window_stuff+323↑p
CODE:004015B6                                         ; maybe_print_flag+E↑p ...
CODE:004015B6
CODE:004015B6 arg_0           = dword ptr  8
CODE:004015B6 arg_4           = dword ptr  0Ch
CODE:004015B6 arg_8           = dword ptr  10h
CODE:004015B6
CODE:004015B6                 enter   0, 0
CODE:004015BA                 mov     eax, [ebp+arg_0]
CODE:004015BD                 add     eax, [ebp+arg_4]
CODE:004015C0                 leave
CODE:004015C1                 retn    8
CODE:004015C1 add             endp
CODE:004015C1
CODE:004015C4
CODE:004015C4 ; =============== S U B R O U T I N E =======================================
CODE:004015C4
CODE:004015C4 ; Attributes: bp-based frame
CODE:004015C4
CODE:004015C4 subtract        proc near               ; CODE XREF: more_window_stuff+33F↑p
CODE:004015C4                                         ; maybe_print_flag+22↑p ...
CODE:004015C4
CODE:004015C4 arg_0           = dword ptr  8
CODE:004015C4 arg_4           = dword ptr  0Ch
CODE:004015C4
CODE:004015C4                 enter   0, 0
CODE:004015C8                 mov     eax, [ebp+arg_0]
CODE:004015CB                 mov     ecx, [ebp+arg_4]
CODE:004015CE                 sub     eax, ecx
CODE:004015D0                 leave
CODE:004015D1                 retn    8
CODE:004015D1 subtract        endp
CODE:004015D1
CODE:004015D4
CODE:004015D4 ; =============== S U B R O U T I N E =======================================
CODE:004015D4
CODE:004015D4 ; Attributes: bp-based frame
CODE:004015D4
CODE:004015D4 multiply        proc near               ; CODE XREF: more_window_stuff+35B↑p
CODE:004015D4                                         ; maybe_print_flag+44↑p ...
CODE:004015D4
CODE:004015D4 arg_0           = dword ptr  8
CODE:004015D4 arg_4           = dword ptr  0Ch
CODE:004015D4
CODE:004015D4                 enter   0, 0
CODE:004015D8                 mov     eax, [ebp+arg_0]
CODE:004015DB                 imul    eax, [ebp+arg_4]
CODE:004015DF                 nop
CODE:004015E0                 leave
CODE:004015E1                 retn    8
CODE:004015E1 multiply        endp
CODE:004015E1
CODE:004015E4
CODE:004015E4 ; =============== S U B R O U T I N E =======================================
CODE:004015E4
CODE:004015E4 ; Attributes: bp-based frame
CODE:004015E4
CODE:004015E4 divide          proc near               ; CODE XREF: more_window_stuff+377↑p
CODE:004015E4                                         ; maybe_print_flag+33↑p
CODE:004015E4
CODE:004015E4 arg_0           = dword ptr  8
CODE:004015E4 arg_4           = dword ptr  0Ch
CODE:004015E4
CODE:004015E4                 enter   0, 0
CODE:004015E8                 mov     eax, [ebp+arg_0]
CODE:004015EB                 mov     ecx, [ebp+arg_4]
CODE:004015EE                 xor     edx, edx
CODE:004015F0                 div     ecx
CODE:004015F2                 leave
CODE:004015F3                 retn    8
CODE:004015F3 divide          endp
```

HV19.17 Unicode Portal

Buy your special gifts online, but for the ultimative gift you have to become admin.

Solution

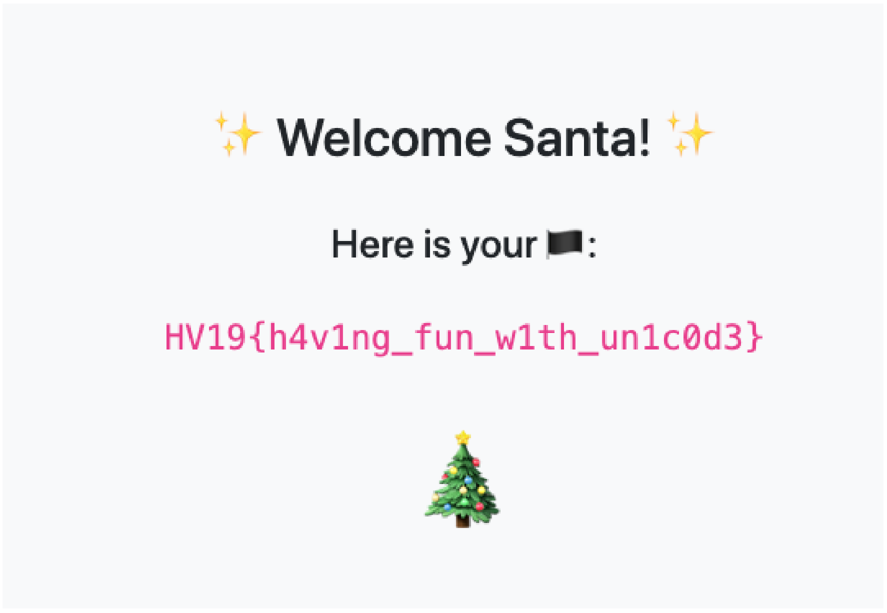
For this challenge, we get access to a website. After registration and log in, we are able to see a page presenting various christmas-themed unicode emojis. Moreover, we are able to take a look at parts of the websites source code. Finally, there is an admin section, to which we do not have access. From the isAdmin check in the source code, we can deduct that we need to log in as user santa to access this part of the page, which most likely contains the flag.

Of course, creating a user named santa fails, because it already exists. So let's dig a bit deeper into the provided code. One thing that quickly caught my attention is, that the sql queries (more particular the username comparison) for checking whether a user already exists, and doing a log in differ:

- The isUsernameAvailable function compares the username using the BINARY (exact case sensitive match) keyword:
    - `... WHERE LOWER(username) = BINARY LOWER('".$usr."')`
- The verifyCreds function compares them using a simple assertion
    - `... WHERE username='".$usr."'"`

Moreover, the SQL statement in the registerUser function contains a very interesting clause (`... ON DUPLICATE KEY UPDATE password='".$pwd."'`). By combining all these findings, I came to the conclusion that it should be possible to register a user with a name that is not equal to santa using exact case sensitive matching (via BINARY keyword), but equal to santa using a simple comparison (= operator). In this case, the ON DUPLICATE KEY trigger would overwrite the password of the santa user, allowing me to log in and get the flag.

A quick google search revealed that this should be easily possible by using characters with accent (e.g. german a Umlaut-A) in santa (see this StackOverflow Post). By registering a user named sànta, I was able to set the admin password. Afterwards, I was able to log in as santa and get the flag.



✨ Welcome Santa! ✨

Here is your 🏴:

HV19{h4v1ng_fun_w1th_un1c0d3}

🎄

As any other hacker could overwrite the password in the meantime, it is important to quickly obtain the flag. :)

Flag: HV19{h4v1ng_fun_w1th_un1c0d3}

HV19.18 Dance with me

Santa had some fun and created todays present with a special dance. this is what he made up for you:

```
096CD446EBC8E04D2FDE299BE44F322863F7A37C18763554EEE4C99C3FAD15
```

Dance with him to recover the flag.

Solution

When we extract the zip file, we get an iOS binary (Mach-O universal binary with 3 architectures), but probably not an iOS app, because we are missing a lot of resources (e.g. plist files). Opening the file up in IDA confirms this assumption. The binary only consists of a few functions. When executed, it reads the flag (maximum 32 bytes) from stdin, and passes it to the dance function, which most likely does some sort of encryption. Afterwards, the ciphertext bytes are printed to stdout. Based on this observation, we can assume that the string we received with the challenge description (31 bytes) is probably our encrypted flag.

My next step was to pinpoint the cipher algorithm. The dance_block function quickly caught my attention, because it contains some suspicious constants (0x61707865, 0x3320646E, 0x79622D32, 0x6B206574). A quick google search reveals, that these constants are part of the Salsa20 cipher.

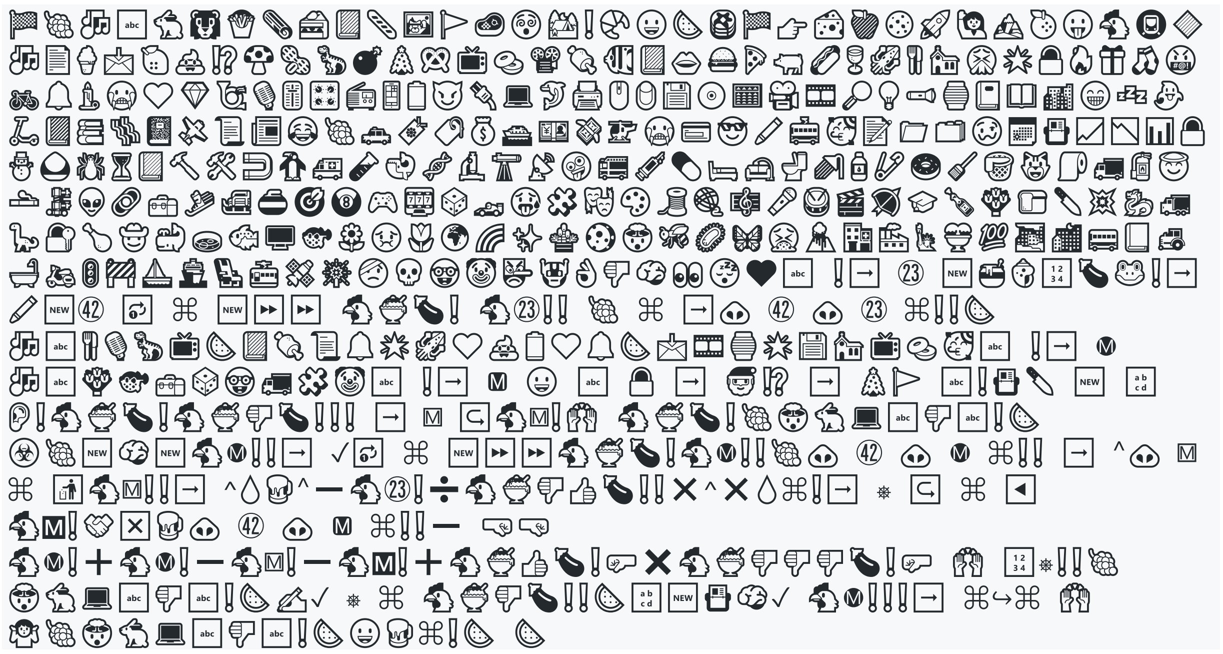
Now we only need to find the key and iv. Most likely, these are passed along with the plaintext to the dance (encryption) function. This happens in 0x100007E2C. Apart from the user supplied input, this function takes two more parameters. One of them (probably the iv), is passed as an immediate value and looks like this: 0x11458fe7a8d032b1. The other one (our key), is copied from the data section starting at 0x100007F50.

With this information, I was able to create a small C# which decrypts the input and prints the flag:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HV1918
{
    class Program
    {
        static void Main(string[] args)
        {
            var encrypted =
StringToByteArray("096CD446EBC8E04D2FDE299BE44F322863F7A37C18763554EEE4C99C3FAD15");
            var salsa = new Salsa20();
            var iv = "11458fe7a8d032b1";
            var decryptor =
salsa.CreateDecryptor(StringToByteArray("0320634661b63cafaa76c27eea00b59bfb2f7097214fd04cb257ac2904efee
46"),
                StringToByteArray(iv));
            var output = new byte[encrypted.Length];
            decryptor.TransformBlock(encrypted, 0, encrypted.Length, output, 0);

            Console.WriteLine(Encoding.ASCII.GetString(output));

            Console.ReadLine();
        }

        public static byte[] StringToByteArray(string hex)
        {
            return Enumerable.Range(0, hex.Length)
                        .Where(x => x % 2 == 0)
                        .Select(x => Convert.ToByte(new string(hex.Substring(x, 2).ToArray()), 16))
                        .ToArray();
        }
    }
}
```

Flag: HV19{Danc1ng_Salsa_in_ass3mbly}

HV19.19 🎅



Solution

For this challenge, we get an obscure sequence of emojis, without any further description. After a while, I discovered the emoji-based programming language emojicode. Our emoji sequence contains the exact starting sequence of an emojicode program (🏁🍇), so I decided to give it a try. Unfortunately, the repo does not offer any simple conversion tools (e.g. emojicode to JS), so I have to dig deeper. After cloning the github repo, I was able to compile an executable from our "source code". When executing this file, I got the following prompt:

🔒 ➡️ 🎅⁉️ ➡️ 🎄🚩

Entering a random word (e.g. flag) results in the following output: 😲 Program panicked: 👎. Probably, we need to enter the correct password so the program prints the flag. When taking a closer look at the prompt, I got the feeling that this password could just be a single emoji. As the number of available emojis is still quite limited, I decided to perform a brute force attack and downloaded a list of allowed unicode emojis from a website and created a small script using pwntools:

```
#!/usr/bin/python
# coding=utf-8
from pwn import *
import json

with open ('emojis.json') as json_file:
    emojis = json.load(json_file)

    for emoji in emojis:
        p = process('./ch19')
        print(p.recvuntil('🔒 ➡️ 🎅⁉️ ➡️ 🎄🚩 '))
        p.sendline(emoji.encode('utf-8'))
        output = p.readall()

        if not "👎" in output:
            print("Done: " + emoji)
            print(output)
            break
```

After a while, the script printed out the flag. The solution (🔑) was quite obvious. 😜

```
🔒 ⍈ 🎅⁉ ⍈ 🎄🏳
[+] Receiving all data: Done (28B)
[*] Process './ch19' stopped with exit code 0 (pid 13015)
Done: 🔑

HV19{*<|:-)____\o/____;-D}
```

Flag: HV19{*<|:-)\o/;-D}

HV19.20 i want to play a game

Santa was spying you on Discord and saw that you want something weird and obscure to reverse?

your wish is my command.

Solution

For this challenge, we get a PS4 game. To my own surpise, mighty IDA was able to load this file without any issues. The binary only contains a few functions. All the interesting stuff seems to happen inside the main function:

First, the function calculates the MD5 sum of the PS4 system software /mnt/usb0/PS4UPDATE.PUP. This has to be f86d4f9d2c049547bd61f942151ffb55. If this matches, the program performs two nested loops, which probably calculate the decrypted flag. The outter loop opens the PS4 system software, seeks to a certain position (starting from 0x1337) and reads a single byte. The inner loop, XORs some byte array (most likely the encrypted flag) from the data section (starting at 0x300) with the byte read from the PS4UPDATE.PUB file. When both loops are done, the resulting byte array (presumably our flag), is sent via the network.s

Based on this findings, we should be able to get the flag by finding the correct system software (based on the md5-hash) and reconstructing the decryption algorithm. A google search shows, that this particular system update is a common PS4 jailbreak (MiraHEN). I downloaded the update file, and created the following python script to get the flag:

```python
#!/usr/bin/python
scrambled_flag =  map(chr,[0xCE, 0x55, 0x95, 0x4E, 0x38, 0xC5, 0x89, 0xA5, 0x1B, 0x6F,
0x5E, 0x25,
                0xD2, 0x1D, 0x2A, 0x2B, 0x5E, 0x7B, 0x39, 0x14, 0x8E, 0xD0, 0xF0, 0xF8,
                0xF8, 0xA5])

flag = scrambled_flag
file_offset = 0x1337

with open('./505Retail.PUP', 'rb') as f:
    while file_offset != 0x1714908:
        f.seek(file_offset)
        key = f.read(len(scrambled_flag))

        newflag = ""
        count = 0
        for char in flag:
            newflag += chr(ord(char) ^ ord(key[count]))
            count += 1

        flag = newflag
        file_offset += 0x1337

print(flag)
```

Flag: HV19{C0nsole_H0mebr3w_FTW}

HV19.21 Happy Christmas 256

Santa has improved since the last Cryptmas and now he uses harder algorithms to secure the flag.

This is his public key:

```
X: 0xc58966d17da18c7f019c881e187c608fcb5010ef36fba4a199e7b382a088072f
Y: 0xd91b949eaf992c464d3e0d09c45b173b121d53097a9d47c25220c0b4beb943c
```

To make sure this is safe, he used the NIST P-256 standard.

But we are lucky and an Elve is our friend. We were able to gather some details from our whistleblower:

Santa used a password and SHA256 for the private key (d)

- His password was leaked 10 years ago
- The password length is the square root of 256
- The flag is encrypted with AES256
- The key for AES is derived with pbkdf2_hmac, salt: "TwoHundredFiftySix", iterations: 256 * 256 * 256

Phew - Santa seems to know his business - or can you still recover this flag?

```
Hy97Xwv97vpwGn21finVvZj5pK/BvBjscf6vffm1po0=
```

Solution

For this challenge, we need to find santas password and recover the flag. As the challenge description already mentions, the flag is encrypted with AES256 and a special key derivation method. Without knowing the password, this is impossible to break within the given timeframe. However, we have some more information.

1. Santas password was leaked 10 years ago and is 11 characters long: As rockyou.txt came out approximately 10 years ago, my first guess was to check all 11 character entries of rockyou.
2. Santa uses the SHA256 of the SAME password, as his private key (d) of his ECC key
3. We also know santas public ECC key (x & y)

Using this information, we could be able to find the correct password. To find the correct entry of rockyou.txt, we try to construct valid ECC keys using the SHA256 of each of the 11-character long entries as private key. If one of them is valid, we found santas password and only need to perform the AES decryption using the same password.

I automated the whole process, using the following python script:

```python
#!/usr/bin/python3

from Crypto.PublicKey import ECC
from Crypto.Cipher import AES
import hashlib
import base64

potential_passwords = []

with open ('./rockyou.txt') as f:
    passwords = f.readlines()
    potential_passwords = list(filter(lambda x: len(x.strip()) == 16, passwords))


print("Found %d potential passwords" % len(potential_passwords))

santas_password = ""
```

```python
for password in potential_passwords:
    x = 0xc58966d17da18c7f019c881e187c608fcb5010ef36fba4a199e7b382a088072f
    y = 0xd91b949eaf992c464d3e0d09c45b173b121d53097a9d47c25220c0b4beb943c
    d = hashlib.sha256(password.strip().encode('utf-8')).hexdigest()
    try:
        private_key = ECC.construct(curve='NIST P-256', d=int(d,16), point_x =x, point_y =y)
        santas_password = password.strip()
        break
    except:
        continue

print("Found santas password: %s" % santas_password)

encrypted_flag = "Hy97Xwv97vpwGn21finVvZj5pK/BvBjscf6vffm1po0="

def decrypt(encrypted, key, salt, rounds):
    encrypted_bytes = base64.decodebytes(encrypted.encode('utf-8'))
    key = hashlib.pbkdf2_hmac('sha256', key.encode('utf-8'), salt, rounds)
    cipher = AES.new(key, AES.MODE_ECB)
    decrypted = cipher.decrypt(encrypted_bytes)
    return decrypted

flag = decrypt(encrypted_flag, santas_password, b'TwoHundredFiftySix', 256 * 256 * 256)
print(flag)
```

Santas password is: `santacomesatxmas`

In theory, we would be also to find the password by directly trying the AES-decryption with all 11-character long entries of rockyou.txt, but due to the huge number of rounds in the key derivation step, this would take significantly longer.

Flag: HV19{sry_n0_crypt0mat_th1s_year}

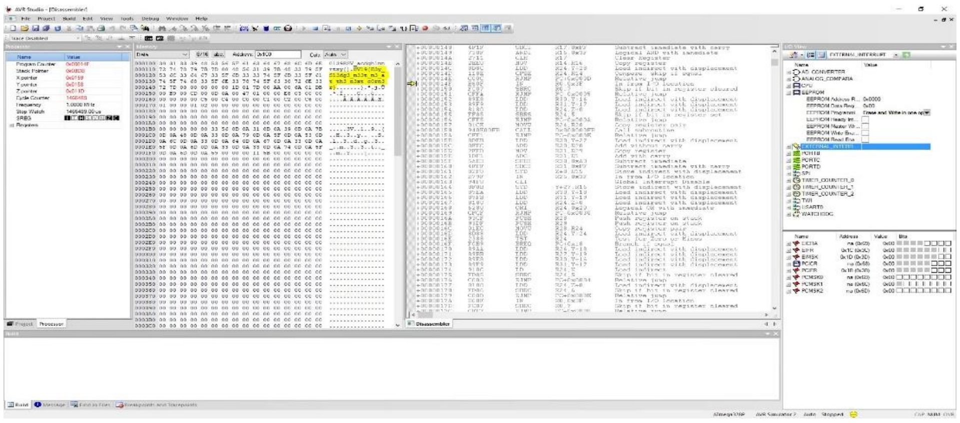
HV19.22 The command ... is lost

Santa bought this gadget when it was released in 2010. He did his own DYI project to control his sledge by serial communication over IR. Unfortunately Santa lost the source code for it and doesn't remember the command needed to send to the sledge. The only thing left is this file: thecommand7.data

Santa likes to start a new DYI project with more commands in January, but first he needs to know the old command. So, now it's on you to help out Santa.

Solution

For this challenge, we get a hex-dump of an Arduino program. As I am not too familiar with the process of reversing Arduino roms, my first approach was to emulate it in AVR Studio. As santas gadget was released in 2010, my first guess for the MCU was ATMega328p, which worked out pretty well. I left the program running for several seconds, and then simply paused it. I opened the subview for inspecting different memory locations, and to my surprise, I simply found the flag on the stack:



Still, I wanted to get some sort of idea what is going on. Therefore, I reconfigured my IDA to handle the Arduino MCU and loaded the hex file. Starting from 0x2be, there are four functions which load certain characters from a predefined set (flag characters, stored in the data section) and stores them in some other location. These sequences of lds and sts instructions seem to build the flag-string, which I found on the stack by emulating the program.

Flag: HV19{H3y_Sl3dg3_m33t_m3_at_th3_n3xt_c0rn3r}

HV19.23 Internet Data Archive

Today's flag is available in the Internet Data Archive (IDA).

Solution

For this challenge, we get a website which allows us to download classic hackvent challenges in a personalized, password protected zip file. Moreover, there is an option to download the flag-challenge, but it is not enabled before 2020. My first guess was to tamper some request parameters, to download a zip file with the flag, but the input validation seems to be pretty strong. After a while, I realized that the /tmp (http://whale.hacking-lab.com:23023/tmp) folder of the application is browsable and contains some interesting files. Apart from the zip archives created by other users, I found a phpinfo.php file, as well as a very old zip archive from the user santa (http://whale.hacking-lab.com:23023/tmp/Santa-data.zip). Downloading the file shows that it contains the flag file. Still we are not able to extract it because we do not know the password.

For each zip-file we create, we get a unique, 12 character long alphanumeric password. This seems to be too long for a pure brute force attack. After a while of googling around, I stumbled across an interesting article about cracking the IDA Pro Installer (https://devco.re/blog/2019/06/21/operation-crack-hacking-IDA-Pro-installer-PRNG-from-an-unusual-way-en/). The IDA Pro installer passwords in this article follow the same pattern as the web application in this challenge. Moreover, the title of this challenge (Internet Data Archive) is an acronym for IDA. Therefore, I decided to reimplement the password cracking algorithm shown in the article in PHP 7.4.1 (taking the same version as mentioned in the phpinfo file). To speed up the cracking process, my password generator only outputs the potential zip-passwords, which get piped into john later on. The script looks as follows:

```php
<?php
$chars = "abcdefghijkmpqrstuvwxyzABCDEFGHJKLMPQRSTUVWXYZ23456789";
for ($seed = 0; $seed <= 1577098880; $seed += 1)
{
    srand($seed);
    $pw = "";
    for($i=0; $i<12; ++$i)
    {
        $key = rand(0, 53);
        $pw = $pw . $chars[$key];
    }
    print $pw . "\n";
}
?>
```

Using this approach, I was able to crack the password within a few seconds:

```
manuel@ManuelErika:/mnt/d/Dokumente/hackvent-2019/23$ php crack.php | john --stdin
santa.hash
Using default input encoding: UTF-8
Loaded 1 password hash (ZIP, WinZip [PBKDF2-SHA1 256/256 AVX2 8x])
Will run 8 OpenMP threads
Press Ctrl-C to abort, or send SIGUSR1 to john process for status
Kwmq3Sqmc5sA     (Santa-data.zip/flag.txt)
1g 0:00:01:01  0.01619g/s 70300p/s 70300c/s 70300C/s suKcApykm6ST..Ekjreg8U85fm
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

Flag: HV19{Cr4ckin_Passw0rdz_like_IDA_Pr0}

HV19.24 ham radio

Elves built for santa a special radio to help him coordinating today's presents delivery.

Solution

For the final challenge, we get a nexmon-patched Broadcom firmware. Using Ghidra, I was able to load the firmware (arm cortex, 32 byte, little endian) and get some reasonable output. The first thing which caught my attention was a base64-encoded string, which translates to "Roses are red, Violets are blue, DrSchottky loves hooking ioctls, why shouldn't you?". Looking at the xrefs, we can see that this string is used in the function located at 0x058dd8. This function seems to be particularly interesting. Although the binary is stripped and many calls to functions are undefined, I was able to find out a couple of things.

If the correct ioctl hooks are called, this function triggers several things. One of them, is probably printing the base64 encoded message we found before (0xcafe hook). Another one, would perform some xor-based decryption. This could be our way to get the flag. By taking a closer look, we can see that the encrypted flag (23 bytes starting from 0x58e94) with the bytes stored at (0x58e8c). I tried to perform this decryption using a simple python script, but it did not work.

After staring at the disassembly for some more time, I discovered that our potential key (0x58ec) could be overwritten with some data coming from 0x800000 (first 23 bytes, see call at 0x58e66). However, this address is not defined in our binary. A google search reveals, that this section is usually reserved for the ROM section of the firmware. After browsing around the seemoo-lab GitHub Repos, I was able to download rom.bin from https://github.com/seemoo-lab/bcm_misc/blob/master/bcm43430a1/rom.bin. I adapted my script to use the first 23 bytes of the rom file as decryption key, and finally got a flag:

```python
#!/usr/bin/python3

key = []
xored_flag = []

with open('./rom.bin', 'rb') as keyfile:
    key = keyfile.read(23)

with open('./brcmfmac43430-sdio.bin', 'rb') as ramfile:
    ramfile.seek(0x58e94)
    xored_flag = ramfile.read(23)


flag = ""
for (k, e) in zip(key, xored_flag):
    flag += chr(k ^ e)

print(flag)
```

Flag: HV19{Y0uw3n7FullM4Cm4n}

HV19.H1 Hidden One

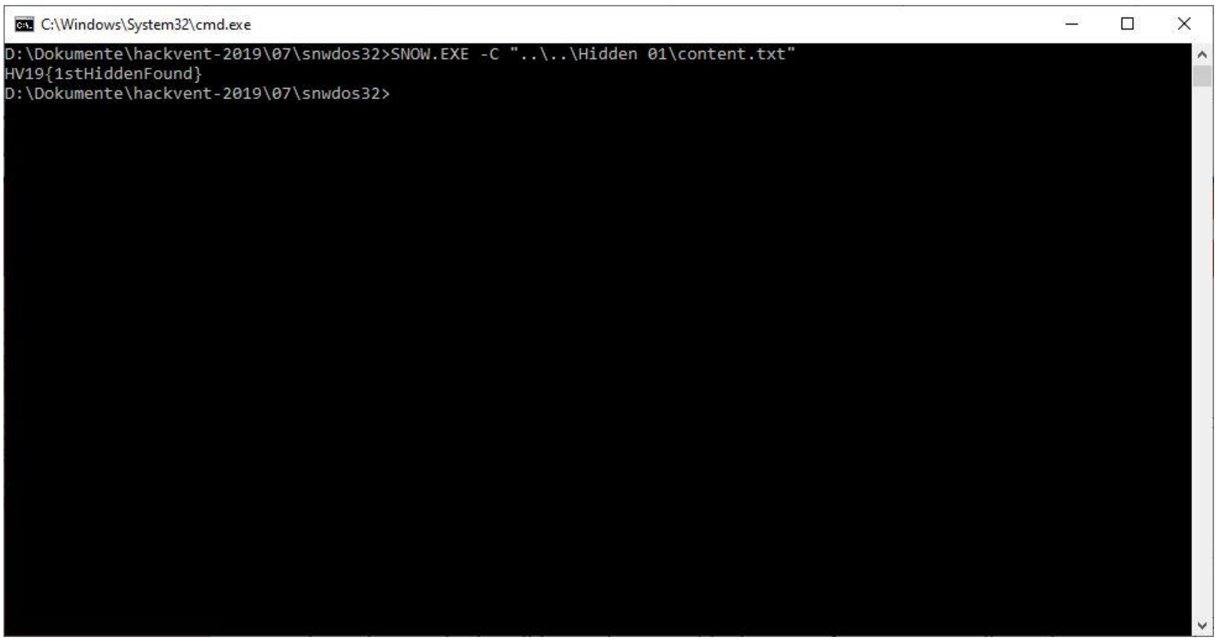
Sometimes, there are hidden flags. Got your first?

Solution

The first hidden flag was released along day #6, so it is highly likely that it's hidden within this challenge. Something that quickly caught my attention was the strange text on the bottom of the challenge description:

```
Born: January 22
Died: April 9
Mother: Lady Anne
Father: Sir Nicholas
Secrets: unknown
```

After taking a closer look, I saw that each line contains a series of trailing whitespaces (tabs and spaces). Maybe this is some sort of whitespace steganography. A quick Google search directed me to the tool Snow. With the help of this program, I was able to retrieve the flag:



```
D:\Dokumente\hackvent-2019\07\snwdos32>SNOW.EXE -C "..\..\Hidden 01\content.txt"
HV19{1stHiddenFound}
D:\Dokumente\hackvent-2019\07\snwdos32>
```

Flag: HV19{1stHiddenFound}

HV19.H2 Hidden Two

Again a hidden flag.

Solution

This hidden flag was released along with day 07. Consequently, I assumed that the hidden flag will be part of this day's challenge. One thing that quickly caught my attention was, that the filename of the video of day 07 was not a valid UUID. This was very suspicious, as all the other files so far followed this common pattern. Also, the filename (3DULK2N7DcpXFg8qGo9Z9qEQqvaEDpUCBB1v) looks like it could be an encoded flag. After quite some time of fiddling around with different encodings and classic ciphers, I discovered that this is just the base58-encoded version of the flag.

Flag: HV19{Dont_confuse_0_and_O}

HV19.H3 Hidden Three

Not each quote is compl

Solution

This hidden challenge was released together with day 11, so it should be related to this challenge. Doing a port scan on the host of the REST Service (whale.hacking-lab.com), shows that one more port is open (17). When connecting to this port via netcat, I got back a single character.

```
manuel@ManuelErika:~$ ncat whale.hacking-lab.com 17
r
```

I tried a few more times, but still got the same results, so I decided to move on for now. In the evening, I tried to connect to this port again, and to my own surprise the server returned a different character. Maybe that is all part of some really long-running hidden challenge.

To prove my point, I created a small cronjob, which periodically queries the endpoint and logs the resulting character to a file. Indeed, after about one day, I was able to use my log output to obtain the flag.

Flag: HV19{an0ther_DAILY_fl4g}

HV19.H4 Hidden Four

Solution

This hidden challenge was released along with day 14. After solving the challenge of day 14, I noticed that the flag has a very weird format and seems to contain a regex:

```
HV19{s@@jSfx4gPcvtiwxPCagrtQ@,y^p-za-oPQ^a-
z\x20\n^&&s[(.)(..)][\2\1]g;s%4(...)%"p$1t"%ee}
```

Another thing that caught my attention, was the sentence Only perl can parse Perl! in the data section of the perl script. Maybe, the whole flag is also a heavily obfuscated perl script. I pasted the flag in a file and executed it:

```
manuel@LAPTOP-A4T7BTE7:/mnt/c/Users/zamet/hackvent-2019/Hidden 04$ perl test.pl
Squ4ring the Circle
```

Flag: HV19{Squ4ring the Circle}