SFI20 CTF Writeups

Contents

all_quiet **3**

astral_projection **4**

byte_matching **5**
Expected approach . . . 5
Example solution . . . 5

competing_standards **6**

elegant_weapons **8**

elegant_weapons_2 **9**

heavyweight **11**
Description . . . 11
Objective . . . 11
Hints . . . 11
Example Solution . . . 11

png **12**

huge_file **13**

important_clues **14**
Description . . . 14
Objective . . . 14
Hints . . . 14
Example Solution . . . 14

message_from_space **15**

sequential_logic **16**
Expected approach . . . 16

sfinux **17**

roundcube **18**

all_quiet

The flag is hidden in an HTML comment.

astral_projection

User is given a matroska formatted video file. When playing it in a semi-advanced video player such as mpv or vlc one can notice an additional, non-default audio track. After selecting it and listening closely, one can hear the flag being read.

byte_matching

The user is given three files: file1.bin, file2.bin and intersection. Upon inspecting the files with a tool like https://iamkate.com/code/binary-file-viewer/, it turns out that the two binary files are rather large and filled with gibberish, while the third file is empty. Searching for the flag manually or with CTRL+F proves ineffective - there is no fragment that directly spells anything resembling the flag format.

Expected approach

The key hint is the name of the empty file. The flag is indeed hidden within the intersection of the two binary files (long binary sequences). Bytes that are the same in both files spell out the flag: sFi20ctf{eDgsLpAnbA}.

Step one Calculate the intersection of the two binary files **Step two** Convert it into text format and read the flag

Example solution

```
def get_intersection(file1_path, file2_path):

    try:

        f1 = open(file1_path, 'rb') #opening in binary reading mode
        f2 = open(file2_path, 'rb')

        file1_data = f1.read()
        file2_data = f2.read()

        length = min(len(file1_data), len(file2_data))

        intersection = bytearray()

        for i in range(length): #reading while there is content to read in both files
            if (file1_data[i] == file2_data[i]):
                intersection.append(file1_data[i])

        f1.close()
        f2.close()

        return bytes(intersection).decode()

    except FileNotFoundError as error: #check for incorrect path to file(s)
        print(f"Error: {error}")
        return None
```

competing_standards

User gets a big binary file and a pdf with a VM specification they need to implement. After running the binary file with VM written by them, it

- does sanity checks to make sure it was implemented correctly,
- asks the user some questions about SFI and one project euler question (the answer for every one can be read from the stack),
- spits out JPEG image containing the flag.

example solution:

```
#include <stdio.h>
#include <stdint.h>
#include <err.h>

#define $(a,b) case(a):{b; break;}
#define N(a,b) $(a, b; ip++)
#define B(n,o) $(n, S[s-2]=S[s-1] o S[s-2]; s--; ip++)
#define I int32_t
#define U uint8_t

I ip, s, c, S[1<<16], C[1<<16], d;
U D[1<<16];

int
main(void)
{
  d=fread(D, 1, 1<<16, fopen("flag.bin", "r")), ip=s=c=0;

  while (ip < d) {
    switch (D[ip]) {
      $(0, S[s++]=(D[ip+4]<<24)|(D[ip+3]<<16)|(D[ip+2]<<8)|D[ip+1]; ip += 5);
      N(1, s--);
      N(3, I t=S[s-1]; S[s-1]=S[s-2]; S[s-2]=t);
      B(4, -);
      B(5, +);
      B(6, *);
      B(7, /);
      B(8, ^);
      B(9, <<);
      B(10, >>);
      N(11, putchar(S[--s]); fflush(stdout));
      $(12, S[s++]=getchar(); ip++);
      $(13, I a=S[--s];I b=S[--s];I c=S[--s]; ip=b==c?a:ip+1; S[s++]=c; S[s++]=b);
      $(14, I a=S[--s];I b=S[--s];I c=S[--s]; ip=b!=c?a:ip+1; S[s++]=c; S[s++]=b);
      $(15, I a=S[--s];I b=S[--s]; ip=b<0?a:ip+1; S[s++]=b);
```

```
      $(16, I a=S[--s];C[c++]=ip+1; ip=a);
      $(17, ip = S[--s]);
      $(18, ip = C[--c]);
      N(19, S[s]=S[s-1]; s++);
      $(20, I a=S[--s]; ip=!s?a:ip+1);
      $(21, I a=S[--s]; ip=s?a:ip+1);
      N(22, I a=S[--s]; I b=S[--s]; D[b]=(U)a%256);
      N(23, I a=S[--s]; S[s++]=D[a]);
      N(24, s--); /* ignore CTF stack */
    default:
      errx(1, "unknown op %d", D[ip]);
    }
  }

  return 0;
}
```

assuming this vm is saved as `vm.c`, to see the solution user could use shell pipes:

```
$ printf '2005\nyes\n6\n21124\nno\n' | tcc -run vm.c | \
  # simulate writing answers          to the vm
  tail -c +729 > /tmp/a.jpg && sxiv /tmp/a.jpg
  # skip questions              view result
```

the CTF stack even yields a hint mentioning pipes.

implementation details will be made public here after sfi20 ends.

elegant_weapons

Files for both elegant_weapons and elegant_weapons2 are FASL-compiled sbcl common lisp image dumps. User can recognise that by running `strings`.

the fasl file stores the flag as bytes + a random number added to every one of them at compile time. The easiest way to find the flag is to call `(sfi20-ctf-flag "dear-user-please-pass-this-value-to-win")`. This constant can be found by running `strings` on `elegant-1.fasl`.

elegant__weapons__2

Expected solution: * running $ strings elegant.fasl one can pick up that it was compiled with `SBCL` version [version-number] (probably 2.2.9.debian); strings like `DEFUN SFI20-CTF-FLAG`, `SFI-FLAG` can also be recognised.

- after opening an sbcl REPL one can then `(load "elegant-2.fasl")`, and try looking for, or using the before-mentioned values. the following session can show how user can find information about the fasl.

```
$ sbcl --noinform --load elegant-2.fasl
* (apropos "sfi")
[...]
SFI20-CTF-FLAG (fbound)
[...]
* (describe 'sfi20-ctf-flag)
COMMON-LISP-USER::SFI20-CTF-FLAG
  [symbol]

SFI20-CTF-FLAG names a compiled function:
  Lambda-list: (X)
  Derived type: (FUNCTION (T) (VALUES LIST &OPTIONAL))
  Source file: (...)

* (sfi20-ctf-flag "hello?")

debugger invoked on a TYPE-ERROR @535D2C87 in thread
#<THREAD "main thread" RUNNING {10048C8253}>:
  The value
    "hello?"
  is not of type
    INTEGER
  when binding X

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.

(SFI20-CTF-FLAG "hello?")
   source: (F X '"#.(SFI-FLAG)")
[...]
* (sfi20-ctf-flag 42)
(#\Nul . #<FUNCTION (LAMBDA (X) :IN SFI20-CTF-FLAG) {1003C50D9B}>)
* (sfi20-ctf-flag 77)
(#\g . #<FUNCTION (LAMBDA (X) :IN SFI20-CTF-FLAG) {1003C55ACB}>)
```

- sfi20-ctf-flag is a function that xors a number given to it with 1st number of `sfi-flag`. this variable is not available to the user, but contains a list of numbers which are bytes of the flag subsequently xorred with (`sfi-begin`, `sfi-begin`+1, ..., `sfi-begin`+N) - `sfi-begin` is also not available to the user. sfi20-ctf-flag returns the un-xorred byte as a character, and a continuation lambda, which will do the same for the next byte in the flag. the process continues until there is no more bytes in `sfi-flag`. the last continuation lambda yields nil. `solve-2.l` contains a functional way of solving the flag. Because the functions return bytes (with values 0-255) there are only 255 possible "keys" which would be the 1st arg (the subsequent calls must be n+1, n+2, ..., n+m) thus it can be easily bruteforced.

```
(defun try-comb (from)
  (let ((func 'sfi20-ctf-flag)
        (i from)
        (bytes '()))
    (loop
      (let ((res (funcall func i)))
        (if res
            (progn
              (setf bytes (append bytes (list (car res))))
              (setf func (cdr res))
              (setf i (1+ i)))
            (return))))
    (coerce bytes 'string)))

(print
 (loop for i from 0 to 255
       collect (cons i (try-comb i))))
```

heavyweight

Description

You are provided with a trained Convolutional Neural Network (CNN) model (`heavyweight_model.pth`) designed to classify handwritten digits from the MNIST dataset. The model architecture is given in the file `model_blueprint.py`.

However, this CNN holds a secret—inside the weights of its first convolutional layer, a hidden message (the flag) has been encoded as pixel art. Your goal is to extract this hidden flag from the provided model.

Objective

Analyze the provided PyTorch model, extract the weights from the convolutional layer, and visualize them as an 8x8 images to retrieve the hidden flag.

Hints

- Inspecting the weights may reveal interesting patterns.
- Consider visualizing the weights as grayscale images or binary pixel art.

Example Solution

```
from model_blueprint import Heavyweight
import torch
import matplotlib.pyplot as plt

model = Heavyweight()
model.load_state_dict(torch.load("heavyweight_model.pth", weights_only=True))
model.eval()

def extract_and_show_kernels(model):
    model.eval()
    kernels = model.features[0].weight.detach().cpu().numpy()

    fig, axes = plt.subplots(1, 8, figsize=(16, 2))
    for i in range(8):
        kernel = kernels[i, 0]
        axes[i].imshow(kernel, cmap='gray', interpolation='nearest')
        axes[i].set_title(f"Kernel {i}")
        axes[i].axis('off')

    plt.tight_layout()
    plt.show()

extract_and_show_kernels(model)
```

png

The file includes 2 additional, non-standard png chunks: `axOR` and `acTF`. `axOR`'s data is a string describing 4 bytes: `0xa5 0x3a 0x1b 0x29`. After xoring the data from `acTF` with `axOR` bytes in order, the result is png data containing the flag.

huge_file

Attempting to download the linked file shows, that it weights almost 1000TB. It also downloads very slowly...

```
Content-Length: 1000000000000000
Accept-Ranges: bytes
```

Inspecting the headers reveals, that the server supports range requests. Requesting the last 100 bytes reveals the flag.

```
% curl -v -r 999999999999900- localhost:5000 --output -  | xxd
* using HTTP/1.x
> GET / HTTP/1.1
> Host: localhost:5000
> Range: bytes=999999999999900-
> User-Agent: curl/8.11.1
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< Server: Werkzeug/3.1.3 Python/3.13.1
< Date: Sun, 05 Jan 2025 17:38:24 GMT
< Content-Type: application/bytes
< Content-Length: 100
< Accept-Ranges: bytes
< Content-Disposition: inline; filename="flag.txt"
< Connection: close
<
{ [1 bytes data]
100   100  100   100    0     0   3385      0 --:--:-- --:--:-- --:--:--  3448
* shutting down connection #0
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0073 6669 3230 5f63  .........sfi20_c
00000050: 7466 7b44 526b 6a72 4c77 794a 687a 3748  tf{DRkjrLwyJhz7H
00000060: 7234 587d                                r4X}
```

important_clues

Category: Forensics / Machine Learning

Files provided: - `forest_flag_challenge.pkl` (trained Random Forest model)

Description

You are provided with a trained Random Forest model (`forest_flag_challenge.pkl`) that classifies data based on multiple features with seemingly random names. At first glance, the feature names look like meaningless sequences of characters. However, there's a hidden secret waiting to be uncovered.

By carefully inspecting the feature importances of this Random Forest model, you will find that sorting the features by ascending importance reveals a hidden message (the flag). Your task is to extract this secret message encoded within the least important features of the model.

Objective

Load the provided Random Forest model, inspect the feature importances, sort them from least important to most important, and read out the hidden flag from the sorted feature names.

Hints

- Use Python and scikit-learn to load and inspect the Random Forest model.
- Sort the feature importances in ascending order to reveal the hidden message.
- Pay close attention to the least important features.

Example Solution

```
import joblib
import numpy as np

data = joblib.load('important_clues_model.pkl')
model = data['model']
feature_names = np.array(data['feature_names'])

sorted_indices = np.argsort(model.feature_importances_)
sorted_features = feature_names[sorted_indices]

print("Najmniej istotne cechy (flaga):")
for name in sorted_features[:20]:
    print(name)
```

message_from_space

The User is given a .wav file that consists of a series of LOW and HIGH noises.

In order to solve this puzzle, they must map said noises to a string of bits (whether they map LOW noise as 0 and HIGH as 1 or the other way around does not matter as it will only flip the colors of black-and-white image), which, upon a close inspection has the length of 2 prime numbers multiplied by each other. They then can convert said string of bits into 2D array with height and width being said 2 prime numbers, and convert the 0 and 1 to pixels, getting an image with the flag.

sequential_logic

The user is given a file mystery.circ and a zipped archive secret.7z The first file has an unusual extension and the user can only read a bunch XML tags in it after opening in a text or code editor. Trying to unzip the archive is unsuccessful, as it is encrypted.

Expected approach

The extension of mystery.circ suggests that it is a circuit file for Logisim. Logisim is free software for creating sequential circuits, among other things. When the user opens the mystery.circ file in Logisim, a circuit connected to a clock appears on the screen. Its only output is an LED diode. When the user turns on the simulation, the diode starts blinking on and off - once for a longer time, once for a shorter time. The diode actually emits signals in Morse code, which can be read either by running the circuit simulation (step by step or all at once at a chosen clock frequency) and directly noting the characters or by constructing a truth table for the circuit.

1. Download Logisim
2. Open mystery.circ in Logisim
3. Add a clock to the unconnected pin on the last J-k flip-flop
4. Trigger clock cycles by clicking on the clock element, read morse code and decode characters
5. Unzip secret.7z with 7 zip, using password from step three
6. Read the flag from secret.txt

sfinux

Website with the task seems to boot up a Linux machine. Upon closer inspection, it's a RISC-V machine emulated in the browser, using TinyEMU/jslinux. The Linux system boots up X11 with TWM. Xterm and xeyes start with the system.

After browsing the system for a while, at least one thing should stand out: a binary called 'access' in /usr/local/bin/. The binary, after being executed, should just output 'nope'. It also happens to have a manpage. `man access` only contains information about some password, but it's not clear where the password should go.

The system has strace installed. Running the binary with strace reveals, that it first loads a module, and then calls some ioctls on `/dev/secret-0`. The module is called `secret.ko`. While it's possible to analyze the module, it's important to remember that many modules are GPL licensed, because they access GPL symbols. The source code of Linux is linked in the licenses page, the source of the module is in `extra` directory.

The module is not very interesting. It detects a memory-mapped device with `secret,mmio` compatible string. One ioctl returns the size, the other copies the data from the device to the userspace.

Where does the data come from? Probably from the emulator, as it emulates all the devices. While most of the emulator code is compiled, some parts (jslinux.js) are not. Close to the beginning of jslinux.js there is a function called `handle_vm_started`. It calls `write_secret` with user-agent, cookies and the URL passed as a string. The device itself is implemented in C part of the emulator (it's compiled to WASM). It's a very simple device, that just maps data from a buffer allocated in the emulator to IO memory.

You can confirm that the memory indeed contains that data using, for example, `devmem` on addresses that are mapped by the driver (see `/proc/iomem` after `modprobe secret`).

It's possible to analyze the binary and check what exactly it does, (whether it reads cookies or something else), but the easiest guess is cookies. The binary also contains a suspicious string `super_secret_value`. Setting a cookie called `super_secret_value` to password from the manpage, refreshing the page and calling the access application again reveals the flag.

roundcube

This task was created by Centrum Bezpieczeństwa Informacji AGH. Thanks!

https://poczta.sfi.pl/poczta/ links to a phishing page prepared to look like a Roundcube clone. Source code analysis shows, that POST request is handled by mail.php.

The server is configured to show directory listing. By viewing https://poczta.sfi.pl/ you can see ./poczta directory in which file called poczta.zip is contained. The archive contains index.html (as seen in the initial link) and the mail.php file that shows, that the passwords are saved to rezults.txt. https://poczta.sfi.pl/poczta/rezults.txt contains the flag.