Baby SoC (justCTF teaser 2024)

> by FluxFingers

In this easy misc/forensics challenge, we are given only a `flashdump.bin` file and this description:

> We found really funny device. It was broken from the beginning, trust us! Can you help with recovering the truth?

Analyzing the Flashdump

`binwalk`ing over the file doesn't yield any interesting results, but calling `strings` on it shows some ESP32-related symbols. We can assume that this is

a flashdump of an ESP32 microcontroller. Now, how to analyze what's going on here?

I found a nice blog post on reversing ESP32 flash dumps, which recommends

a tool called `esp32-image-parser`. With some patches from the open PRs applied, we can dump the sections of the flashdump:

```
$ ./esp32_image_parser.py show_partitions ../flashdump.bin

reading partition table...
entry 0:
  label      : nvs
  offset     : 0x9000
  length     : 20480
  type       : 1 [DATA]
  sub type   : 2 [NVS]

entry 1:
  label      : otadata
  offset     : 0xe000
  length     : 8192
  type       : 1 [DATA]
  sub type   : 0 [OTA]

entry 2:
  label      : app0
  offset     : 0x10000
  length     : 1310720
  type       : 0 [APP]
  sub type   : 16 [ota_0]

entry 3:
  label      : app1
  offset     : 0x150000
  length     : 1310720
  type       : 0 [APP]
  sub type   : 17 [ota_1]

entry 4:
  label      : spiffs
  offset     : 0x290000
  length     : 1441792
  type       : 1 [DATA]
  sub type   : 130 [SPIFFS]

entry 5:
  label      : coredump
  offset     : 0x3f0000
  length     : 65536
  type       : 1 [DATA]
  sub type   : 3 [COREDUMP]

MD5sum:
972dae2ff872a0142d60bad124c0666b

Done
```

As per the blog post, we can now transform the application sections (only `app0` turned out to matter in our case) to ELF

files:

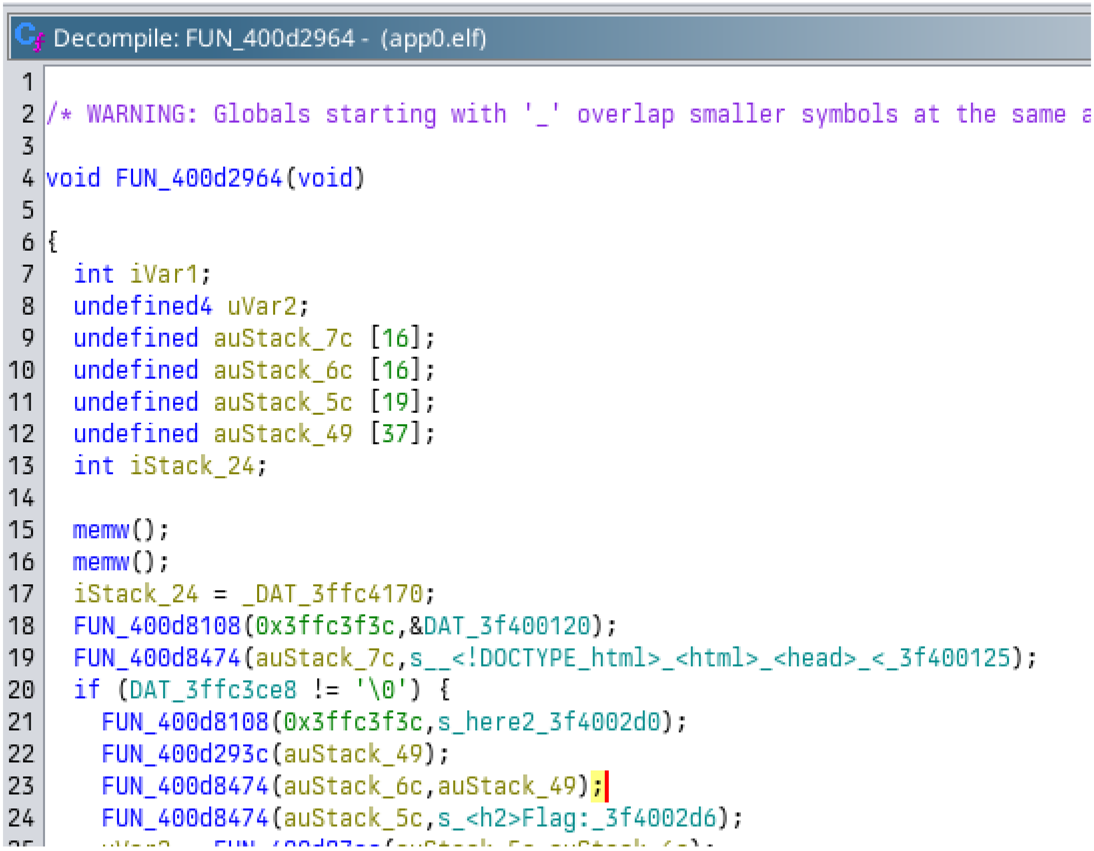
```
./esp32_image_parser.py create_elf ../flashdump.bin -partition app0 -output app0.elf
```

Now we have an ELF file that we can analyze with Ghidra.

Analyzing the Application

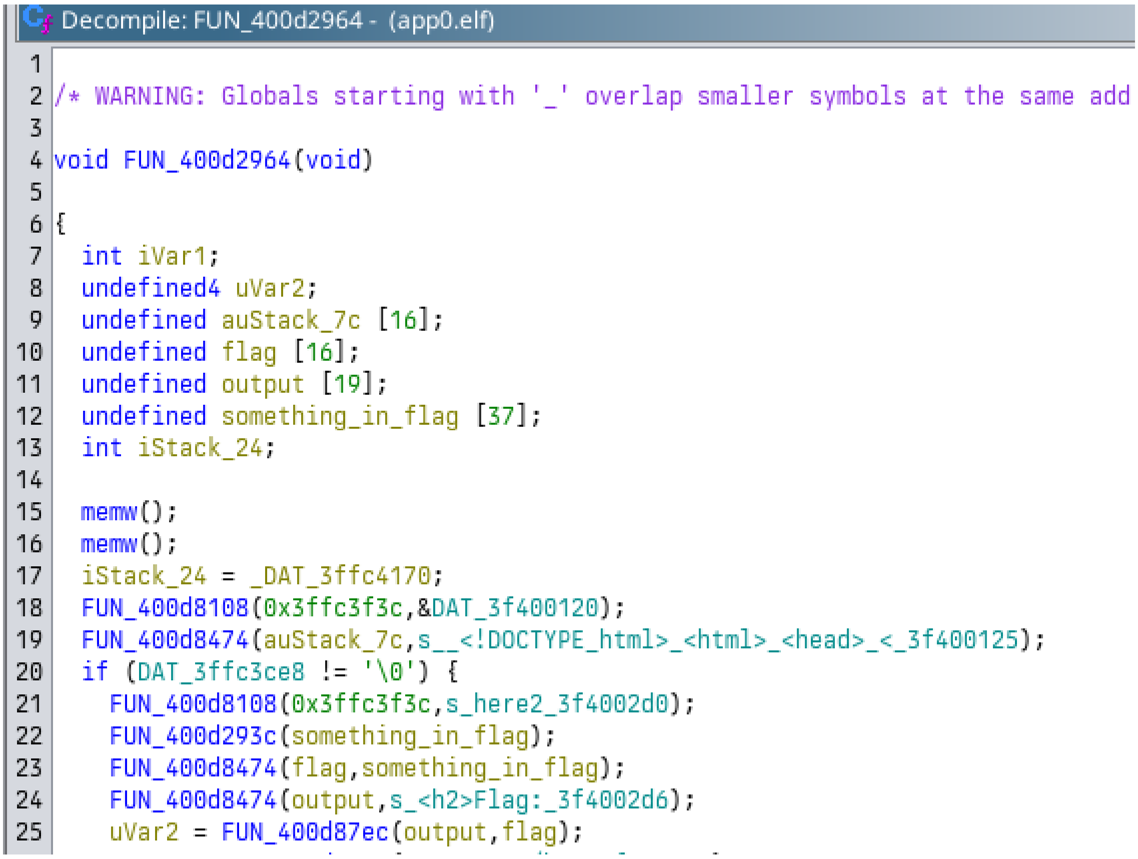
Looking through the strings used in the file, we can quickly find that there's some HTML for displaying the flag. Therefore, this should be a web server

of sorts.



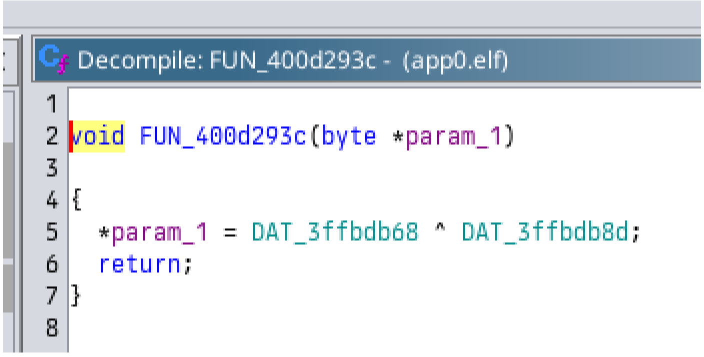
```

/* WARNING: Globals starting with '_' overlap smaller symbols at the same a

void FUN_400d2964(void)

{
  int iVar1;
  undefined4 uVar2;
  undefined auStack_7c [16];
  undefined auStack_6c [16];
  undefined auStack_5c [19];
  undefined auStack_49 [37];
  int iStack_24;

  memw();
  memw();
  iStack_24 = _DAT_3ffc4170;
  FUN_400d8108(0x3ffc3f3c,&DAT_3f400120);
  FUN_400d8474(auStack_7c,s__<!DOCTYPE_html>_<html>_<head>_<_3f400125);
  if (DAT_3ffc3ce8 != '\0') {
    FUN_400d8108(0x3ffc3f3c,s_here2_3f4002d0);
    FUN_400d293c(auStack_49);
    FUN_400d8474(auStack_6c,auStack_49);
    FUN_400d8474(auStack_5c,s_<h2>Flag:_3f4002d6);
```

Looking at where the stuff gets written to, we can quickly find the part that should write the flag:



```

/* WARNING: Globals starting with '_' overlap smaller symbols at the same add

void FUN_400d2964(void)

{
  int iVar1;
  undefined4 uVar2;
  undefined auStack_7c [16];
  undefined flag [16];
  undefined output [19];
  undefined something_in_flag [37];
  int iStack_24;

  memw();
  memw();
  iStack_24 = _DAT_3ffc4170;
  FUN_400d8108(0x3ffc3f3c,&DAT_3f400120);
  FUN_400d8474(auStack_7c,s__<!DOCTYPE_html>_<html>_<head>_<_3f400125);
  if (DAT_3ffc3ce8 != '\0') {
    FUN_400d8108(0x3ffc3f3c,s_here2_3f4002d0);
    FUN_400d293c(something_in_flag);
    FUN_400d8474(flag,something_in_flag);
    FUN_400d8474(output,s_<h2>Flag:_3f4002d6);
    uVar2 = FUN_400d87ec(output,flag);
```

Analyzing where the values come from, we can find that the flag is computed by XORing two values from the data section into `something_in_flag`:



```

void FUN_400d293c(byte *param_1)

{
  *param_1 = DAT_3ffbdb68 ^ DAT_3ffbdb8d;
  return;
}

```

Performing this XOR in Python gives us the flag:

> `justCTF{you_x0r_me_r1ght_r0und_b4by}`