



Analyse matérielle et exploitation du STM32F405

Rapport technique

Nom de l'équipe : FlashBack

Composition de l'équipe :

- BRAHIMI Massinissa
- LATTARI Rayane
- SZOKE Stefan

Année Universitaire : 2024/2025

SOMMAIRE

SOMMAIRE.....	2
I. Introduction.....	3
II. Analyse matérielle.....	3
III. Analyse logicielle et exploitation.....	4
A. Identification du microcontrôleur.....	4
B. Recherche et consultation du datasheet.....	4
C. Connexion et extraction du firmware.....	4
D. Dump de la mémoire flash.....	6
IV. Analyse du firmware avec ghidra.....	6
A. Chargement du SVD (System View Description).....	7
Chargement de la SRAM.....	10
V. Analyse des commandes TLV.....	14
VI. Scripts et Automatisation.....	15
VII. Conclusion.....	16
VIII. Annexes.....	17

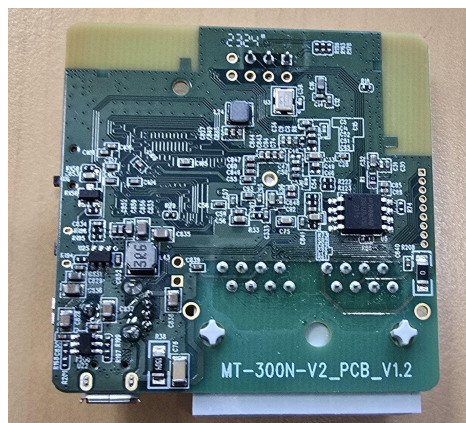
I. Introduction

Ce document présente les analyses et exploitations réalisées dans le cadre de l'examen de sécurité des systèmes embarqués, il est structuré en deux parties :

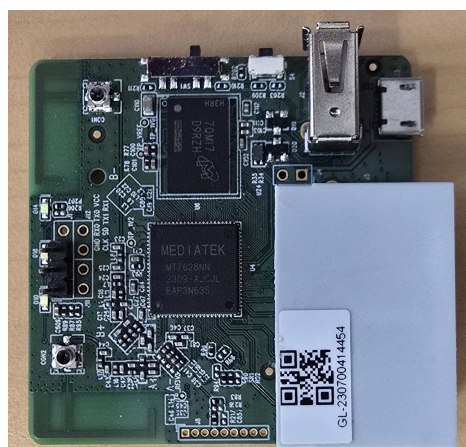
- **Analyse matérielle** : Identification des composants, architecture et stockage du code.
- **Analyse logicielle et exploitation** : Interaction avec le microcontrôleur , recherche de vulnérabilités , analyse et extraction de données

II. Analyse matérielle

Avant de commencer notre analyse matérielle, nous avons pris différentes photos afin de pouvoir identifier les composants.



Photographie de la face supérieure de la PCB



Photographie de la face inférieure de la PCB



Photographie de la face inférieure du boîtier

Nous avons ainsi pu identifier les composants et références suivants :

- Mini Smart Router : GL-MT300N-V2
- PCB : MT-300N-V2_PCB_V1.2
- System on Chip : Mediatek : MT7628NN2309-AJCJL EAP3N635
- Mémoire flash : Winband : 25Q128JVSQ 2316
- Mémoire RAM : Micron : 7QMI7D9RZH

A l'aide des différentes références identifiées, nous avons pu trouvés les datasheets et programming manual correspondants :

- System on Chip : Mediatek :
 Datasheet : https://vonger.cn/upload/MT7628_Full.pdf
 Programming manual :
<https://service-manual.eu/view/1576276/mediatek-ralink-mt7620-programming-manual-523/>
- Mémoire flash : Winband :
<https://www.alldatasheet.com/datasheet-pdf/view-marking/1243793/WINBOND/W25Q128JVSQ.html>
 On observe une lettre de différence entre la référence obtenue et la référence du datasheet. Le désignateur "I" dans W25Q128JVSQ indique que la puce est un dispositif de qualité industrielle avec une plage de température étendue ce qui la distingue de son homologue standard (de qualité commerciale, version sans I). Le datasheet trouvé reste donc pertinent.
- Mémoire RAM : Micron :
<https://www.compel.ru/pdf-items/micron/pn/mt47h128m16rt-25e-it-c/51f5a424321769aa5d09187f3c81b877>
- La référence D9RZH correspond à la référence complète suivante : MT47H128M16RT-25E. Le datasheet trouvé est donc pertinent.

Ces datasheets nous permettent d'obtenir des informations importantes sur les différents composants :

- Le microcontrôleur principal est le MT7628NN2309-AJCJL EAP3N635. Il a une architecture MIPS 24KEc.
- Le code est stocké sur la SPI Flash externe

Pour le STM32F405 on a pu identifier les composants références suivantes :

- Convertisseur de niveau logique : YF08E 14K CVQ0
- Régulateur de tension linéaire : LM 1117 3.3v

A l'aide des différentes références identifiées, nous avons pu trouvés les datasheets correspondants :

- STM32F405 :
[//www.alldatasheet.com/datasheet-pdf/download/1424913/STMICROELECTRONICS/STM32F405.html](http://www.alldatasheet.com/datasheet-pdf/download/1424913/STMICROELECTRONICS/STM32F405.html)
- convertisseur de niveau logique :
<https://www.ti.com/lit/ds/symlink/txs0108e.pdf>
- régulateur de tension linéaire :
<https://www.ti.com/lit/ds/symlink/lm1117.pdf>

Il existe divers moyens d'extraire et de modifier le firmware. On peut notamment utiliser :

- OpenOCD et GDB
- FlashROM

III. Analyse logicielle et exploitation

A. Identification du microcontrôleur

Voir partie “I. Analyse matérielle”.

B. Recherche et consultation du datasheet

Une recherche a été effectuée pour obtenir la documentation technique du *STM32F405* (voir partie I). Celle-ci a permis d'identifier les éléments critiques suivants :

- **Emplacement du firmware** : Mémoire Flash interne
- **Table des interruptions (Vector Table)** : 0x08000000
- **Adresse du pointeur de pile au démarrage** : 0x08000000
- **Adresse du vecteur de réinitialisation** : 0x08000004

C. Connexion et extraction du firmware

L'extraction du firmware a été réalisée en connectant une sonde **ST-Link** au port de debug JTAG de la carte.

L'identification des périphériques connectés a été vérifiée avec la commande :

```
$ lsusb
```

La commande suivante a permis d'obtenir les logs du port utilisé par le microcontrôleur :

```
$ dmesg -w
```

```
[25418.430607] usb 2-2.3: new full-speed USB device number 23 using uhci_hcd
[25418.739822] usb 2-2.3: New USB device found, idVendor=0483, idProduct=5740, bcdDevice= 2.00
[25418.739828] usb 2-2.3: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[25418.739829] usb 2-2.3: Product: Hack Me if you can
[25418.739830] usb 2-2.3: Manufacturer: GotoHack
[25418.739830] usb 2-2.3: SerialNumber: 205037973548
[25418.745491] cdc_acm 2-2.3:1.0: ttyACM0: USB ACM device
[25936.805522] usb 2-2.3: USB disconnect, device number 23
[25940.072272] usb 2-2.3: new full-speed USB device number 24 using uhci_hcd
[25940.385298] usb 2-2.3: New USB device found, idVendor=0483, idProduct=5740, bcdDevice= 2.00
[25940.385304] usb 2-2.3: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[25940.385305] usb 2-2.3: Product: Hack Me if you can
[25940.385306] usb 2-2.3: Manufacturer: GotoHack
[25940.385307] usb 2-2.3: SerialNumber: 205037973548
[25940.389340] cdc_acm 2-2.3:1.0: ttyACM0: USB ACM device
[28220.775855] usb 2-2.3: USB disconnect, device number 24 (Info: device disconnected, state == 0)
[28253.576514] usb 2-2.3: new full-speed USB device number 25 using uhci_hcd
[28253.869333] usb 2-2.3: New USB device found, idVendor=0483, idProduct=5740, bcdDevice= 2.00
```

Le microcontrôleur a été détecté sur le port `/dev/ttyACM0`, ce qui a servi pour la connexion.

Une connexion avec **OpenOCD** a ensuite été établie. L'outil a signalé que le microcontrôleur attendait une connexion via le port **3333** pour **GDB**.

(grâce au serveur gdb lancé sur le port 3333)

```
(kali㉿kali)-[~/.../university/reverse/Hard/examen]
$ openocd -f interface/stlink.cfg -f target/stm32f4x.cfg
Open On-Chip Debugger 0.12.0
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override use 'transport
select <transport>'.
Info : The selected transport took over low-level target control. The results might differ co
mpared to plain JTAG/SWD
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 2000 kHz
Info : STLINK V2J37S7 (API v2) VID:PID 0483:3748
Info : Target voltage: 3.298003
Info : [stm32f4x.cpu] Cortex-M4 r0p1 processor detected
Info : [stm32f4x.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for stm32f4x.cpu on 3333
Info : Listening on port 3333 for gdb connections
Info : accepting 'gdb' connection on tcp/3333
[stm32f4x.cpu] halted due to debug-request, current mode: Thread
xPSR: 0x81000000 pc: 0x08002a06 msp: 0x2001ff70
Info : device id = 0x10076413
Info : flash size = 1024 KiB
Info : flash size = 512 bytes
Info : dropped 'gdb' connection
```

La connexion a été effectuée avec les commandes suivantes:

```
$ gdb-multiarch -q
$ set architecture arm
$ target extended-remote localhost:3333
```

```
(kali㉿kali)-[~/.../university/reverse/Hard/examen]
$ gdb-multiarch -q
(gdb) set arch arm
The target architecture is set to "arm".
(gdb) target extended-remote localhost:3333
Remote debugging using localhost:3333
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x08002a06 in ?? ()
```

D. Dump de la mémoire flash

Avant l'extraction, une consultation du datasheet a permis d'identifier les régions mémoire à cibler :

(<https://www.alldatasheet.com/datasheet-pdf/download/1424913/STMICROELECTRONICS/STM32F405.html>)

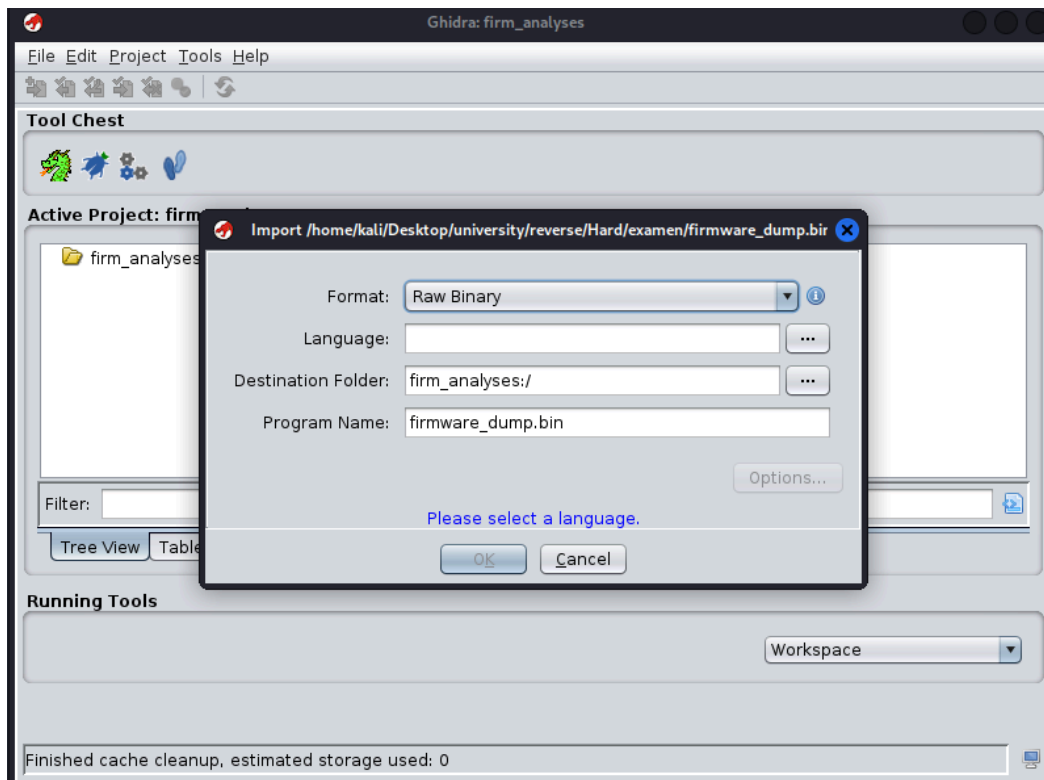
- **Adresse de début** : 0x08000000
- **Adresse de fin** : 0x080FFFFF

L'extraction a été réalisée et un hash **SHA256** a été calculé pour vérifier l'intégrité des données :

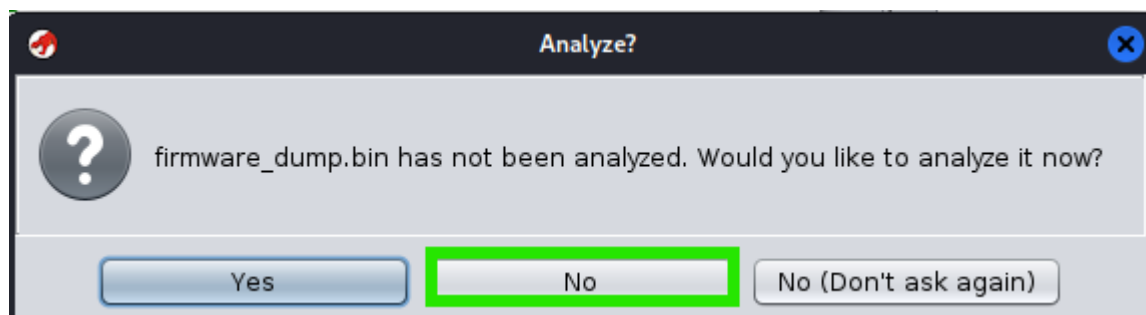
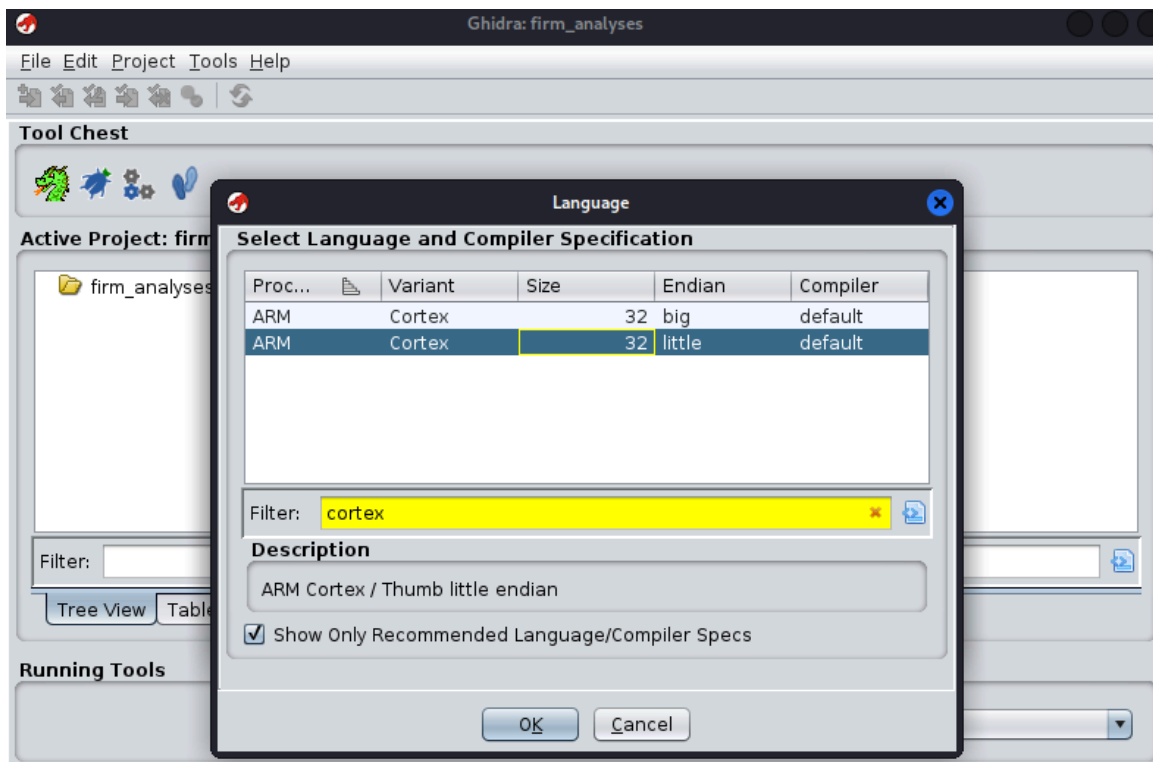
```
0x08002a06 in ?? ()  
(gdb) dump memory firmware_dump.bin 0x08000000 0x080FFFFF  
(gdb) █
```

```
(kali@kali)-[~/.../university/reverse/Hard/examen]  
$ sha256sum firmware_dump.bin  
d0cb70edbd598c25fc01856d08f2ba9ae68b61b87820f94310091433c6ad1be2  firmware_dump.bin
```

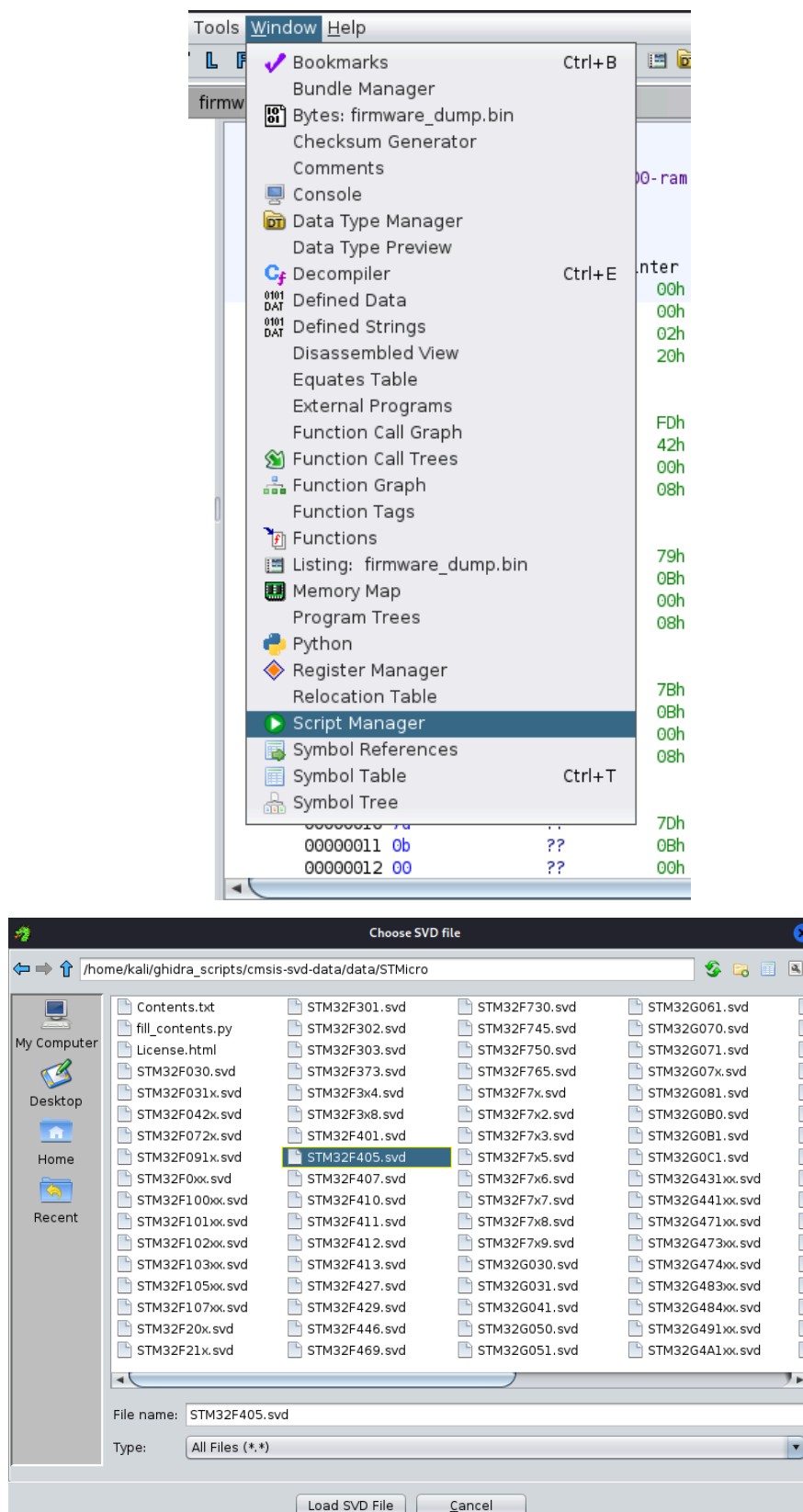
IV. Analyse du firmware avec ghidra

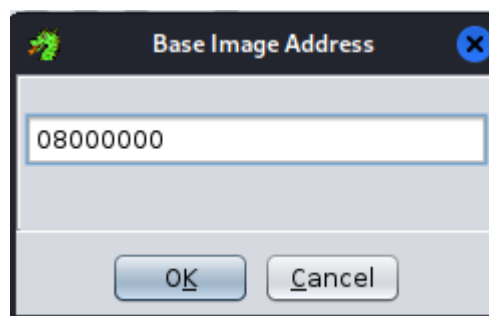
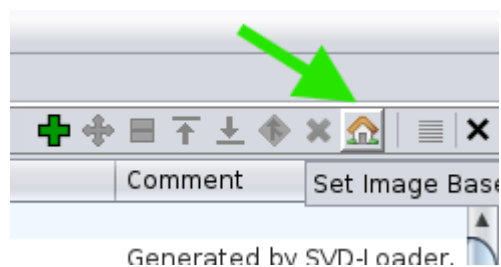
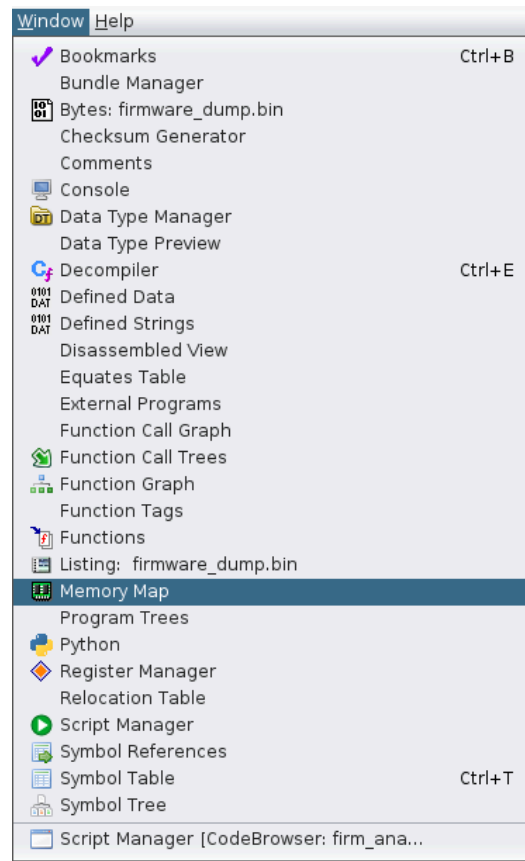


Un projet a été créé sous **Ghidra** pour l'analyse statique du firmware extrait. Cette étape nous a permis d'identifier des vulnérabilités au niveau de la logique ainsi que dans l'implémentation du code.



A. Chargement du SVD (System View Description)





Add Memory Block

Block Name:

Start Addr:

Length:

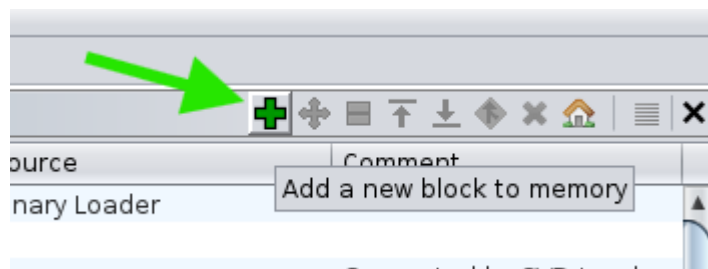
Comment:

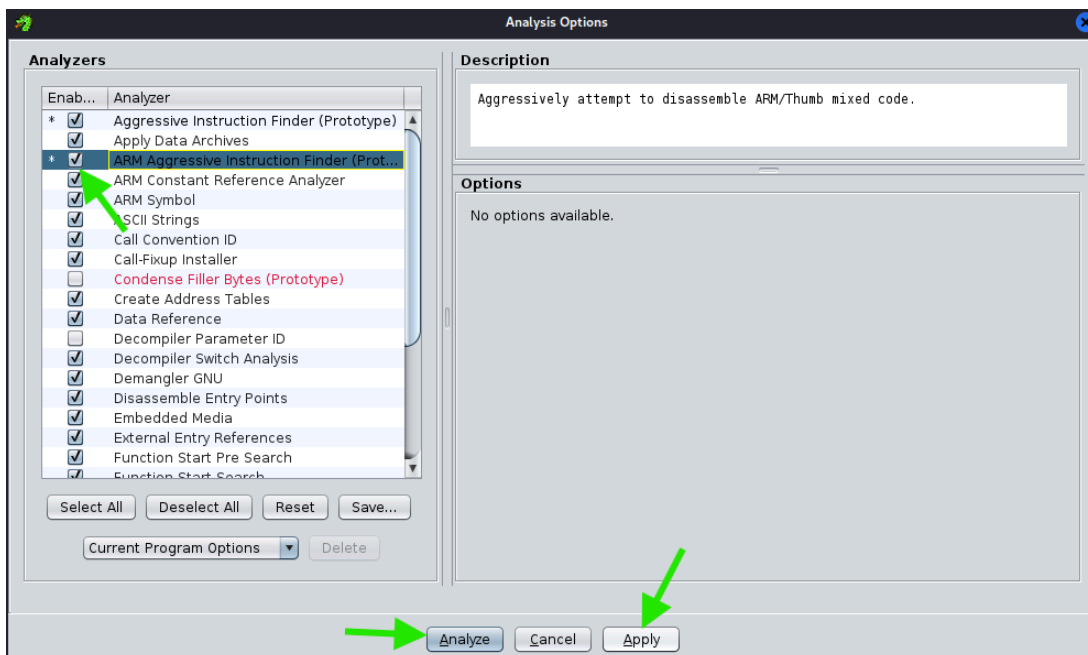
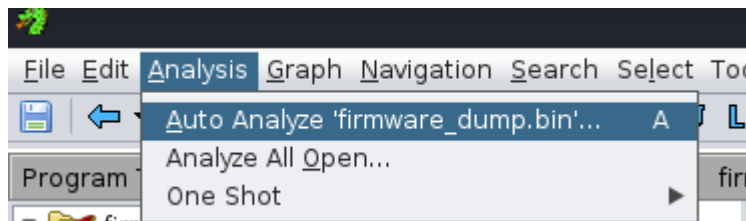
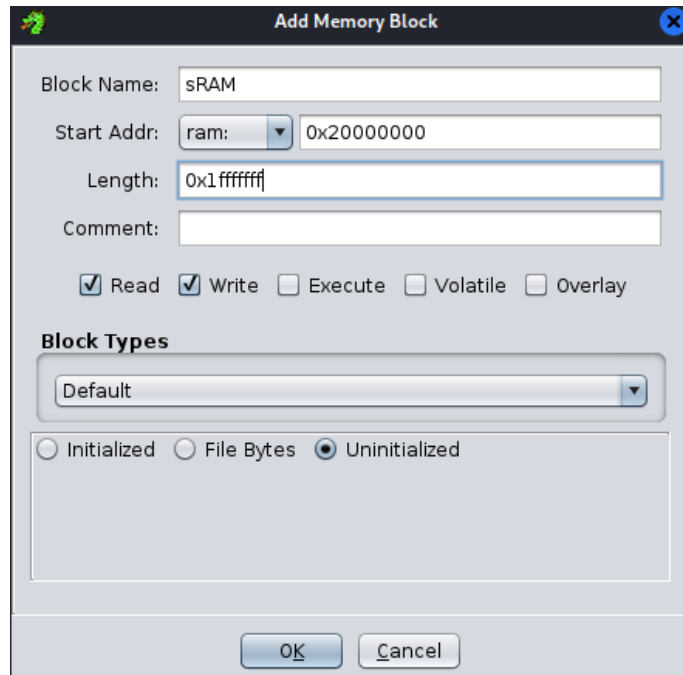
☒ Read ☒ Write ☐ Execute ☐ Volatile ☐ Overlay

Block Types

☐ Initialized ☐ File Bytes ☒ Uninitialized

Chargement de la SRAM







```

1
2 void auth_function(void)
3
4 {
5     byte bVar1;
6     int iVar2;
7     undefined1 password_buffer [32];
8     undefined1 user_input_buffer [64];
9
10    while( true ) {
11        if ( 4 < PTR_DAT_08000b28->field0_0x0 ) {
12            printf(msg_too_many_attempt);
13            return;
14        }
15        printf(PTR_s_Enter_password);
16        read_user_input(user_input_buffer,0x40);
17        load_password(password_buffer);
18        iVar2 = compareString(user_input_buffer,password_buffer);
19        if (iVar2 == 0) break;
20        bVar1 = PTR_DAT_08000b28->field0_0x0;
21        if ((1 < bVar1) && (iVar2 = FUN_080001d0(user_input_buffer,PTR_s_DEBUG123_08000b34), iVar2 == 0)
22            ) {
23            *DAT_08000b20 = 1;
24            *DAT_08000b38 = 1;
25            printf(debug_mode_enabled_access_granted);
26            return;
27        }
28        PTR_DAT_08000b28->field0_0x0 = bVar1 + 1;
29        printf(PTR_s_Access_denied_Try_again);
30    }
31    *DAT_08000b20 = 1;
32    printf(tvl_access_granted);
33    return;
34 }
35

```

Le mot de passe pour entrer en mode debug est : **DEBUG123**

```

FUN_08000a20(auStack_48,0x40);
FUN_080007c4(auStack_68);
iVar2 = FUN_08000830(auStack_48,auStack_68);

```

On constate que le mot de passe est codé sur 7 octet, et qu'il se situe à l'adresse indiquée "DAT_08000800" (adresse pointée par la variable PTR_encoded_password). On trouve les valeurs suivantes:

```

1
2 void load_password(int password_buffer)
3
4 {
5     uint uVar1;
6
7     for (uVar1 = 0; uVar1 < 7; uVar1 = uVar1 + 1) {
8         *(byte *)(password_buffer + uVar1) = *(byte *) (PTR_encoded_password + uVar1) ^ 0x5a;
9     }
10    *(undefined1 *) (password_buffer + 7) = 0;
11    if (PTR_DAT_08000804->field0_0x0 == '\x01') {
12        printf(String_Decrypted_password);
13        printf(password_buffer);
14        printf(String_Entering_TLV_command_mode);
15    }
16    return;
17 }
18

```

02000014	36	??	36h	6
02000015	3f	??	3Fh	?
02000016	2e	??	2Eh	.
02000017	37	??	37h	7
02000018	3f	??	3Fh	?
02000019	33	??	33h	3
0200001a	34	??	34h	4

On code un programme python qui va nous permettre de faire le xor inverse.

```

reveal_pass.py > ...
1  buffer = [0x36, 0x3f, 0x2e, 0x37, 0x3f, 0x33, 0x34]
2
3  passwd = ''
4  for i in range(0, 7):
5      passwd += chr(buffer[i] ^ 0x5a)
6  print(passwd)

```

```
(kali@kali) - [~/.../university/reverse/Hard/pass_finder]
$ python3 reveal_pass.py
letmein
```

```
Access denied. Try again
Access denied. Try again
[DEBUG MODE ENABLED]
Access granted!
Entering TLV command mode ...
04f3...
```

Ainsi le mot de passe est : “**letmein**” afin d’entrer en mode TLV

Pistes essayées :

Pour récupérer le mot de passe nous avons essayé une autre approche se basant sur l’utilisation de gdb en mode debug.

```
1
2 void load_password(int password_buffer)
3
4 {
5     uint uVar1;
6
7     for (uVar1 = 0; uVar1 < 7; uVar1 = uVar1 + 1) {
8         *(byte *) (password_buffer + uVar1) = *(byte *) (PTR_encoded_password + uVar1) ^ 0x5a;
9     }
10    *(undefined1 *) (password_buffer + 7) = 0;
11    if (PTR_DAT_08000804->field0_0x0 == '\x01') {
12        printf(String_Decrypted_password);
13        printf(password_buffer);
14        printf(String_Entering_TLV_command_mode);
15    }
16    return;
17 }
18
```

Dans la fonction permettant de charger le mot de passe on constate qu’il existe un “**if**” permettant de faire un print du mot de passe decoder.

Nous avons tenté 2 approches pour essayer d’afficher le mot de passe.

Approche 1 :

```
LAB_080007d8                                XREF[1]: 080007d8
080007d8 06 2b      cmp     r3,#0x6
080007da f7 d9      bls     LAB_080007cc
080007dc 00 23      movs    r3,#0x0
080007de e3 71      strb    r3,[r4,#0x7]
080007e0 08 4b      ldr     r3,[PTR_DAT_08000804]
080007e2 1b 78      ldrb    r3,[r3,#0x0]=>DAT_20000195
080007e4 01 2b      cmp     r3,#0x1
080007e6 00 d0      beq     LAB_080007ea

LAB_080007e8                                XREF[1]: 080007e8
080007e8 10 bd      pop     {r4,pc}

LAB_080007ea                                XREF[1]: 080007ea
080007ea 07 48      ldr     password_buffer=>s_[DEBUG]_Decrypted_password:...
080007ec ff f7 3e ff  bl      printf
080007f0 20 46      mov     password_buffer,r4
080007f2 ff f7 3b ff  bl      printf
080007f6 05 48      ldr     password_buffer=>s__08004ec8+28,[String_Enteri...
080007f8 ff f7 38 ff  bl      printf
080007fc f4 e7      b       LAB_080007e8
080007fe 00      ??     00h
080007ff bf      ??     BFh
```

En analysant le code assembleur, on se rend compte que la valeur est comparé à **“0x01”** et est placé dans le registre **“r3”**.

On a ainsi essayé de mettre un breakpoint avec gdb, sur l’adresse de l’instruction qui permet de faire la comparaison, c’est à dire de

080007e4 cmp r3, #01

Par la suite, notre idée était de placer dans le registre **“r3”**, la bonne valeur. Cette approche n’a pas abouti.

Approche 2 :

La seconde approche se base également sur l’utilisation d’un breakpoint, mais a la place on mettra directement la bonne valeur, c’est à dire **“#1”** dans la variable.

De cette façon on pourra à nouveau entrer dans le **“if”** et afficher le mot de passe.

V. Vulnérabilité et obtention du flag

On a utilisé dans cette partie le code avec les noms de fonctions et les variables **“wargame.elf”**, car on n’a pas en notre possession le **STM32F405** ainsi que le **st-Link**

Flag :

Dans la fonction **“ProcessTLV”**, nous avons remarqué l’utilisation d’une fonction nommée **“DecryptFlag”**.

Cette fonction fait des opérations de xor sur un tableau à l’adresse 0x20000000.

```
1
2 /* WARNING: Unknown calling convention */
3
4 void DecryptFlag(char *output)
5
6 {
7     uint uVar1;
8
9     for (uVar1 = 0; uVar1 < 0x11; uVar1 = uVar1 + 1) {
10         output[uVar1] = *(byte *) (uVar1 + 0x20000000) ^ 0x5a;
11     }
12     output[0x11] = '\0';
13     return;
14 }
15
```

On retrouve dans l’adresse un tableau de bytes :

20000000	19 0e 1c	uint8_t[...	19h, 0Eh, 1Ch, "!", 12h, "?;*", 15h, ", ?{.
	21 12 3f		
	3b 2a 15 ...		
-	20000000 [0]	19h, 0Eh, 1Ch, '!',	
-	20000004 [4]	12h, '?', ';', '*',	
-	20000008 [8]	15h, ',', '?', '(',	
-	2000000c [12]	'<', '6', '5', '-',	
-	20000010 [16]	'\'', 00h	
	20000012 00	??	00h
	20000013 00	??	00h

Afin d'obtenir le flag, il a suffi de faire l'opération inverse avec le code suivant :

```
1  def decrypt_flag():
2      group0 = [0x19, 0x0E, 0x1C, 0x21]
3      group1 = [0x12, ord('?'), ord(';'), ord('*')]
4      group2 = [0x15, ord(','), ord('?'), ord('(')]
5      group3 = [ord('<'), ord('6'), ord('5'), ord('-')]
6      group4 = [ord(""), 0x00]
7
8      raw_bytes = group0 + group1 + group2 + group3 + group4
9      encrypted = raw_bytes[:17]
10
11     decrypted_bytes = [b ^ 0x5A for b in encrypted]
12
13     flag = ''.join(chr(b) for b in decrypted_bytes)
14     return flag
15
16 if __name__ == '__main__':
17     result = decrypt_flag()
18     print("Flag :", result)
19
```

Code permettant de réaliser l'opération XOR inverse

On obtient ainsi le “flag” suivant :

“CTF{HeapOverflow}”

Vulnérabilité 1 : *Timing attack*

Lors de notre analyse du code, nous avons observé une vulnérabilité de type “*timing attack*”. En effet, dans la fonction “*LoginPrompt*”, on observe une autre fonction appelée “*TimingVulnerableStrcmp*”.

```

void LoginPrompt(void)
{
    uint8_t uVar1;
    int iVar2;
    char decrypted [32];
    char input [64];

    while( true ) {
        if (4 < failed_attempts) {
            SendMessage("Too many failed attempts. Access denied permanently.\r\n");
            return;
        }
        SendMessage("Enter password: ");
        ReceiveMessage(input,0x40);
        DecryptPassword(decrypted);
        iVar2 = TimingVulnerableStrcmp(input,decrypted);
        uVar1 = failed_attempts;
        if (iVar2 == 0) break;
        if ((1 < failed_attempts) && (iVar2 = strcmp(input,"DEBUG123"), iVar2 == 0)) {
            authenticated = '\x01';
            debug_mode = '\x01';
            SendMessage("[DEBUG MODE ENABLED]\r\nAccess granted!\r\n");
            return;
        }
    }
}

```

Capture du code de “LoginPrompt” décompilé sur Ghidra
On étudie ensuite la fonction “*TimingVulnerableStrcmp*”.

```

int TimingVulnerableStrcmp(char *s1,char *s2)
{
    uint uVar1;
    int iVar2;

    for (iVar2 = 0; (uVar1 = (uint) (byte)s1[iVar2], uVar1 != 0 && ((byte)s2[iVar2] != 0));
        iVar2 = iVar2 + 1) {
        if (uVar1 != (byte)s2[iVar2]) {
            return 1;
        }
        HAL_Delay(0x32);
    }
    iVar2 = uVar1 - (byte)s2[iVar2];
    if (iVar2 != 0) {
        iVar2 = 1;
    }
    return iVar2;
}

```

Capture du code de la fonction “*TimingVulnerableStrcmp*” décompilé sur Ghidra

Cette fonction compare 2 chaînes de caractères. Cependant, dès qu’on rencontre un caractère différent, on quitte la fonction. Et dès que les caractères sont identiques, la fonction ajoute un délai de 50ms (**HAL_Delay(0x32)**).

Ainsi on peut tester tous les caractères en mesurant les délais respectifs et dès que l’on tombe sur un caractère avec un délai plus long on sait que l’on a un caractère valide.

On peut ainsi réitérer l’opération jusqu’à obtenir le mot de passe dans son entièreté. On a donc bien la “*timing attack*” vulnérabilité .

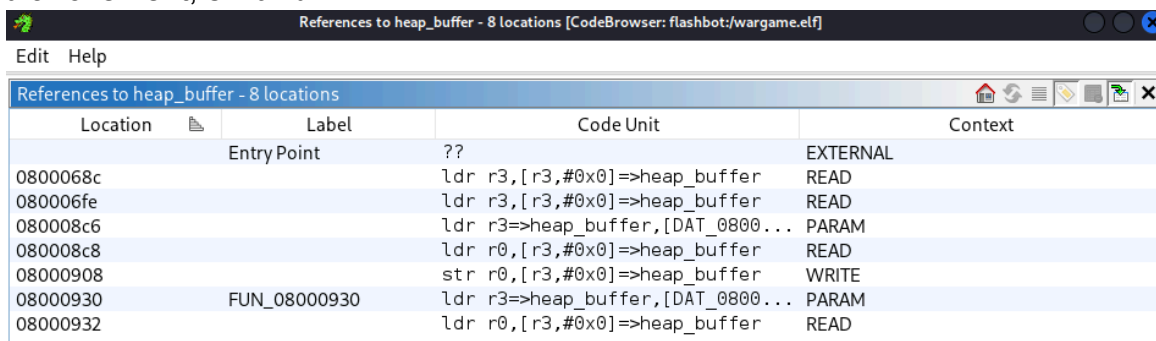
Vulnérabilité 2 : *Heap Buffer Overflow*

Dans “*ProcessTLV*”, on appelle “*memcpy*” avec “*heap_buffer*” en destination sans contrôler sa taille.

```
if (uVar2 == 2) {
    if (heap_buffer != (uint8_t *)0x0) {
        memcpy(heap_buffer,buffer + 3,__size);
        SendMessage("[+] Heap overflow triggered.\r\n");
        return;
    }
    SendMessage("[-] No allocated memory.\r\n");
    return;
}
```

Le programme va copier l'entièreté du contenu du buffer (de taille “*__size*”), sans prendre de précautions sur la taille de ce buffer. Et la fonction “*memcpy*” ne fait pas de vérification supplémentaire pour savoir si elle écrit en dehors des limites du buffer. Ainsi un heap overflow serait possible d'arriver.

Après une analyse approfondie du code, l'exploitation de cette vulnérabilité semble possible. En effet, la seule référence en écriture de cette variable globale se trouve dans la même fonction. Elle est cependant exécutée dans un autre branchement, si “*uVar2 == 1*” :



Location	Label	Code Unit	Context
	Entry Point	??	EXTERNAL
0800068c		ldr r3,[r3,#0x0]=>heap_buffer	READ
080006fe		ldr r3,[r3,#0x0]=>heap_buffer	READ
080008c6		ldr r3=>heap_buffer,[DAT_0800...	PARAM
080008c8		ldr r0,[r3,#0x0]=>heap_buffer	READ
08000908		str r0,[r3,#0x0]=>heap_buffer	WRITE
08000930	FUN_08000930	ldr r3=>heap_buffer,[DAT_0800...	PARAM
08000932		ldr r0,[r3,#0x0]=>heap_buffer	READ

```
if (uVar2 == 1) {
    heap_size = uVar1;
    heap_buffer = (uint8_t *)malloc(__size);
    if (heap_buffer != (uint8_t *)0x0) {
        SendMessage("[+] Memory allocated.\r\n");
        return;
    }
    SendMessage("[-] Allocation failed.\r\n");
    return;
}
```

Il faudrait que la fonction soit exécutée plusieurs fois, une fois avec un message de type 1, et une seconde fois un message de type 2. De cette façon on aura alloué de la mémoire sans l'avoir libérée, et on pourra satisfaire la condition

“if (heap_buffer != (uint8_t *)0x0)”

Et alors on peut avoir un heap overflow en utilisant envoyant une taille de message inférieure en paramètre au premier message, et supérieur au second. Ce paramètre de taille correspondant a “*uVar1*”, qui est initialisé avec la valeur contenue dans les deux octets commençant à l’adresse buffer+1.

```
1
2 /* WARNING: Unknown calling convention -- yet parameter storage is locked */
3
4 void TLV_CommandProcessor(void)
5
6 {
7     uint8_t input [64];
8
9     SendMessage("Entering TLV command mode...\r\n");
10    ReceiveMessage((char *)input,0x40);
11    while (authenticated != '\0') {
12        ProcessTLV(input,0x40);
13    }
14    return;
15 }
16
```

On a le contrôle sur cette valeur car le buffer correspond à l’input qu’on envoie.

La fonction “ReceiveMessage” utilise strncpy pour copier le contenu du buffer global rxBuffer dans le tampon fourni, en réservant le dernier octet pour le caractère de terminaison nul ('\0').

De plus, elle remplace les caractères de saut de ligne (ASCII 10) et de retour chariot (ASCII 13) par des caractères nuls, garantissant ainsi une chaîne propre. On a le contrôle sur cette valeur, car le buffer correspond à l’input qu’on envoie.

On peut alors changer la valeur de “*length*”, sans avoir à se soucier qu’elle se fasse changer ou nettoyer.

VI. Conclusion

Lors de cette semaine nous avons découvert la discipline de la rétro-ingénierie matérielle. Nous avons ainsi pu acquérir les techniques et principes de base utilisés dans cette discipline. Cela nous a permis également de revoir certaines notions vu dans le cours de rétro-ingénierie logicielle (par exemple l’utilisation de Ghidra) tout en découvrant des aspects complètement nouveaux comme la partie électronique.

Le devoir concluant la semaine nous a permis de mettre en pratique les notions vues au cours de la semaine sur un problème plus complet et complexe. Nous avons ainsi pu extraire le firmware du microcontrôleur avant d’étudier son code afin de découvrir de potentielles failles de sécurité.