Reverse Engineering

Sécurité des systèmes embarqués

Clement C. Carole F. Léonard Namolaru

23 mars 2025

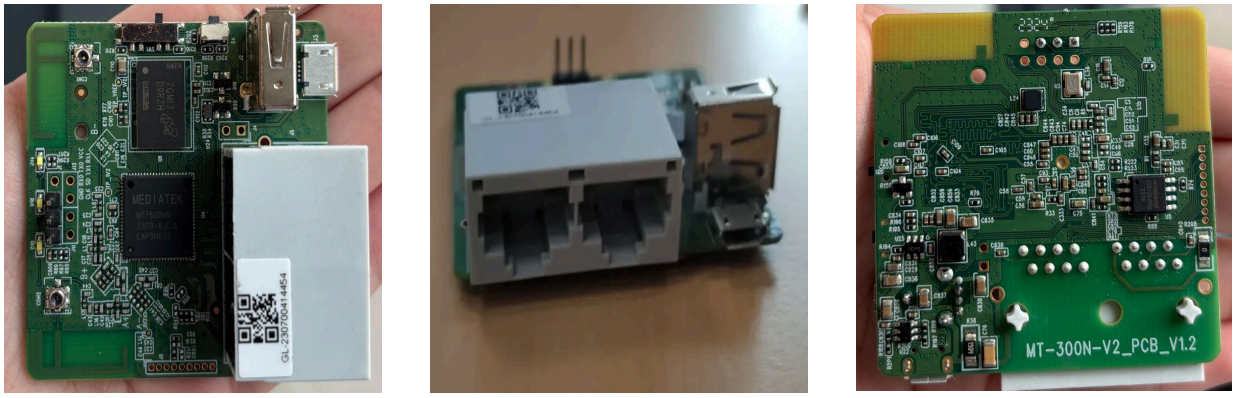
La structure globale du fichier est basée sur un modèle de la galerie des modèles de Google Docs : Fiche de lecture par Reading Rainbow

Sommaire

Analyse matérielle

1. Analyse matérielle du routeur GL.iNet 300M

1.1 Analyse matérielle avec photos



Photos de la carte mère du routeur GL.iNet 300M : vue du dessus, de face et arrière

Ces images présentent la carte électronique du routeur sous différents angles. Nous pouvons identifier :

- Les puces principales (SoC, mémoire Flash et RAM).
- Les interfaces physiques (ports USB, micro-USB, Ethernet).
- Les connecteurs de debug (UART).
- Le blindage métallique, qui protège les composants RF (Radio Fréquence).

1.2 Composants identifiés

- **SoC (System on Chip)** : MediaTek MT7628NN - Processeur principal du routeur, intègre un processeur MIPS 24KEc cadencé à 580 MHz, un module Wi-Fi 2.4 GHz, un contrôleur Ethernet et la gestion de l'interface USB.
- **Mémoire Flash** : Winbond 25Q128JVSQ (128Mb soit 16Mo de NOR Flash) - Contient le firmware du routeur, basé sur OpenWRT.
- **Mémoire RAM** : Micron D9RZH (7QMI7) (64 Mo de DDR2) - Stocke les données en cours d'exécution.
- **Composants RF** :
    - Un module Wi-Fi intégré au SoC permet la transmission sur la bande 2.4 GHz (802.11n).
    - Un blindage métallique recouvre certains circuits pour éviter les interférences électromagnétiques.
    - Une piste d'antenne est présente sur le PCB, assurant l'émission et la réception du signal Wi-Fi.

1.3 Architecture du routeur

- **Processeur et gestion du réseau**
    - Le SoC MediaTek MT7628NN gère l'ensemble des fonctionnalités du routeur, incluant le Wi-Fi, l'Ethernet et l'USB. Il intègre également un switch Ethernet pour la gestion des ports réseau.
- **Stockage et exécution du système**
    - Le firmware OpenWRT est stocké dans la mémoire Flash Winbond de 16 Mo.

    - Il est chargé en RAM Micron D9RZH 64 Mo DDR2 lors de l'exécution.
    - Un port UART (J12) permet un accès direct au terminal du routeur pour du debug ou du flashage de firmware.
- **Connectivité et interfaces**
    - Wi-Fi 2.4 GHz : Permet de créer un point d'accès ou un répéteur.
    - Ports Ethernet WAN/LAN : Assurent la connectivité filaire en 10/100 Mbps.
    - Port USB Type-A : Peut être utilisé pour connecter un stockage externe ou un modem 4G.
    - Port micro-USB : Fournit l'alimentation en 5V/1A.

1.4 Stockage du code

Le GL.iNet 300M Mini Smart Router stocke son firmware OpenWRT dans une mémoire Flash NOR externe Winbond W25Q128JVSIQ, connectée au SoC MediaTek MT7628NN via un bus SPI. Au démarrage, le SoC exécute le bootloader U-Boot depuis cette mémoire Flash, qui initialise le matériel et charge le noyau Linux en RAM DDR2 Micron D9RZH pour exécution. Le protocole utilisé pour la communication entre la Flash et le SoC est SPI NOR Flash Read/Write.

1.5 Extraction et modification du firmware

Le routeur GL.iNet 300M offre plusieurs méthodes pour extraire ou modifier le firmware OpenWRT :

- **Via l'interface web OpenWRT** (LuCI) : Permet de mettre à jour facilement le firmware depuis l'interface graphique.
- **Par connexion SSH** : En accédant au système de fichiers et aux partitions MTD via terminal (cat /dev/mtdX, scp, etc.).

- **Par la console UART** (port J12) : Accessible physiquement sur la carte, ce port permet d'entrer dans U-Boot pour interrompre le démarrage, lancer une mise à jour via TFTP, ou reprogrammer la mémoire.
- **Via un programmeur SPI** : Si le firmware est corrompu, il est possible de dessouder ou clipper la puce Flash Winbond W25Q128 et d'en extraire le contenu avec un lecteur SPI (comme CH341A) et des outils comme flashrom.

Analyse, recherche de bugs et exploitation

1. Analyse du PCB et extraction du firmware

1.1 Analyse du PCB

Composants visibles et leurs références

- **Microcontrôleur principal** : STM32F405RGT6, son architecture est basée sur le cœur ARM Cortex-M4, avec unité de calcul en virgule flottante (FPU) et instructions DSP.
- **Mémoire externe probable** : Circuit CMS à gauche du MCU.
- **Connecteur USB-A**
- **Connecteur SWD** (5 broches) : VCC, SWCLK, GND, SWIO, NRST.
- **Quartz** : Oscillateur à boîtier métallique.
- **Régulateur de tension** : Probable AMS1117 (boîtier SOT-223).
- **Composants passifs** : Résistances, condensateurs CMS.

1.2 Localisation des éléments critiques

Stockage du firmware

- Le firmware est stocké dans la mémoire Flash interne du microcontrôleur STM32F405RGT6, qui possède 1 Mo de Flash intégrée.
- Adresse de base : `0x08000000`

Table des interruptions (Vector Table)

Au moment du reset, la table des vecteurs d'interruptions est localisée par défaut à l'adresse 0x00000000. Cependant, une fois le système démarré, le firmware peut modifier cette adresse via le registre VTOR (Vector Table Offset Register) pour pointer, par exemple, vers 0x08000000 (début de la Flash interne).

- Adresse par défaut : **0x00000000**
- Adresse après relocation : **0x08000000**

Adresse du pointeur de pile au démarrage

Le pointeur de pile initial (MSP) est la première entrée de la table des vecteurs. Il est donc lu à l'adresse :

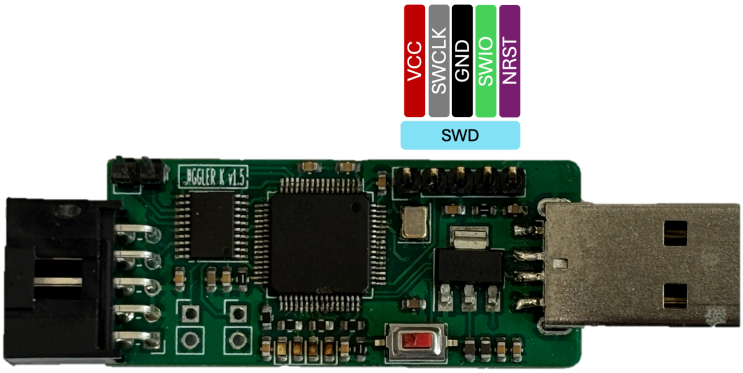
- 0x00000000 juste après le reset (avant modification du VTOR)
- Cette valeur est copiée dans le registre SP (Stack Pointer) par le processeur.

Adresse du vecteur de réinitialisation

Le vecteur de réinitialisation est la deuxième entrée de la table des vecteurs, immédiatement après le pointeur de pile. Il est lu à l'adresse :

- 0x00000004
- Il contient l'adresse de la fonction Reset_Handler, qui est exécutée immédiatement après un reset.

1.3 Connexion JTAG et dump du firmware



git clone https://github.com/openocd-org/openocd

openocd -f openocd/tcl/interface/stlink.cfg -f openocd/tcl/target/stm32f4x.cfg

```
┌──(namolaru㉿kali)-[~]
└─$ openocd -f openocd/tcl/interface/stlink.cfg -f openocd/tcl/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 "dapdirect_swd". To override use 'transport select <transport>'.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : STLINK V2J37S7 (API v2) VID:PID 0483:3748
Info : Target voltage: 3.332609
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : stlink_dap_op_connect(connect)
Info : SWD DPIDR 0x2ba01477
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: 0x08002d3c msp: 0x2001ff70
Info : device id = 0x10076413
Info : flash size = 1024 KiB
Info : flash size = 512 bytes
Info : dropped 'gdb' connection
```

gdb-multiarch

(gdb) set architecture arm

(gdb) target extended-remote localhost:3333

(gdb) dump memory firmware_dump_after_update.bin 0x08000000 0x08010000

(gdb) quit

```
(gdb) set architecture 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.
0x08002d3c in ?? ()
(gdb) dump memory firmware_dump_after_update.bin 0x08000000 0x08010000
(gdb) quit
```

sha256sum firmware_dump_after_update.bin

```
┌──(namolaru㉿kali)-[~]
└─$ sha256sum firmware_dump_after_update.bin
54cea34810a2a227f654754617cf3660e8991e3229e134a28fe6af68d8266794  firmware_dump_after_update.bin
```

1.4 Dump de la ram

(gdb) dump binary memory dump_ram.bin 0x20000000 0x20020000

```
┌──(myenv)─(clem㉿kali)-[~/Documents/reverse/exam/exo2]
└─$ gdb-multiarch
GNU gdb (Debian 16.2-2) 16.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) set architecture 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.
0x08002d3e in ?? ()
(gdb) dump memory firmware_dump_bcp.bin 0x08000000 0x08010000
(gdb) dump memory firmware_dump_bcp.bin 0x02000000 0x04000000
(gdb) dump binary memory dump_ram.bin 0x20000000 0x20020000
(gdb) dump memory dump_ram_no_binary.bin 0x20000000 0x20020000
(gdb) exit
A debugging session is active.
```

1.5 Analyse du SVD (System View Description)

Identification des périphériques mémoire et registres importants

Le fichier SVD utilisé pour cette analyse est le fichier officiel fourni pour le microcontrôleur STM32F405RGT6, dont le lien est référencé dans la bibliographie. Ce fichier, structuré en XML, décrit l'ensemble des périphériques mémoire mappés du système ainsi que leurs registres internes, voici les plus importants :

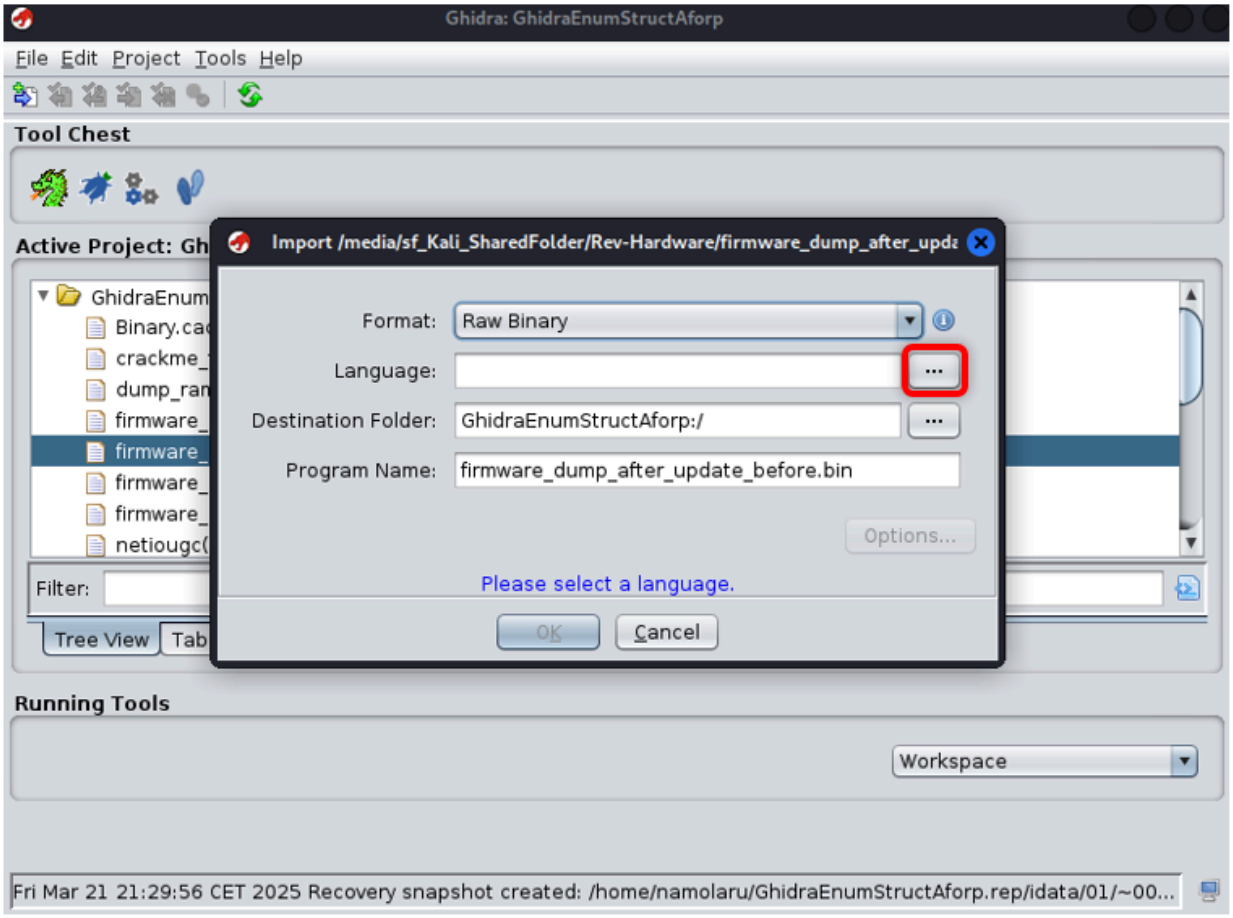
- **RCC (Reset and Clock Control)**
  - CR - Contrôle de l'oscillateur principal et PLL
  - PLLCFGR - Configuration de la PLL
  - CFGR - Configuration de l'horloge système
  - CIR - Contrôle des interruptions liées à l'horloge
  - AHB1ENR, APB1ENR, APB2ENR - Activation des horloges pour les périphériques
- **GPIOA à GPIOI (Ports d'entrée/sortie)**
  - MODER - Configuration des modes des broches (entrée, sortie, alternatif...)
  - OTYPER - Type de sortie (push-pull/open-drain)
  - OSPEEDR - Vitesse de la broche
  - PUPDR - Pull-up/pull-down configuration
  - IDR - Lecture des entrées
  - ODR - Écriture des sorties
  - BSRR - Mise à 1 ou à 0 d'une broche
  - LCKR - Verrouillage de configuration
  - AFR[0], AFR[1] - Fonctions alternatives
- **USART1, USART2, USART3, UART4, UART5 (Communication série)**
  - SR - Status (drapeaux de transmission, réception...)
  - DR - Données à envoyer ou reçues
  - BRR - Baud rate (vitesse de transmission)

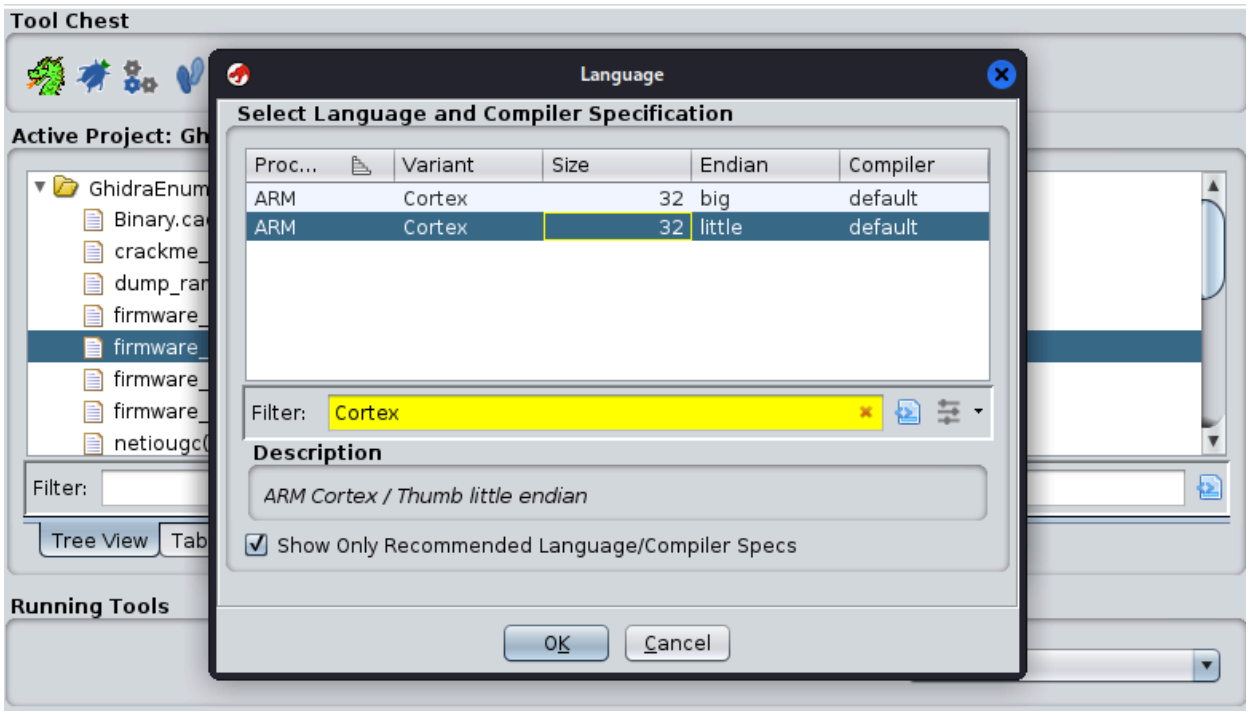
    - CR1, CR2, CR3 - Contrôle de la configuration UART
- **SPI1, SPI2, SPI3 (Interface SPI)**
    - CR1, CR2 - Configuration (mode maître/esclave, vitesse, etc.)
    - SR - Statut SPI
    - DR - Registre de données
- **I2C1, I2C2, I2C3 (Interface I^2C)**
    - CR1, CR2 - Contrôle
    - SR1, SR2 - Statuts et événements
    - DR - Données
    - CCR - Contrôle de la vitesse
    - TRISE - Temps de montée
- **USB_OTG_FS / USB_OTG_HS (Interface USB OTG)**
    - GOTGCTL, GOTGINT - Contrôle général
    - GAHBCFG, GUSBCFG - Configuration du bus et de l'interface USB
    - GRSTCTL, GINTSTS - Reset, statut des interruptions
    - DIEPCTL, DOEPCTL - Contrôle des endpoints (IN/OUT)

2. Configuration de Ghidra

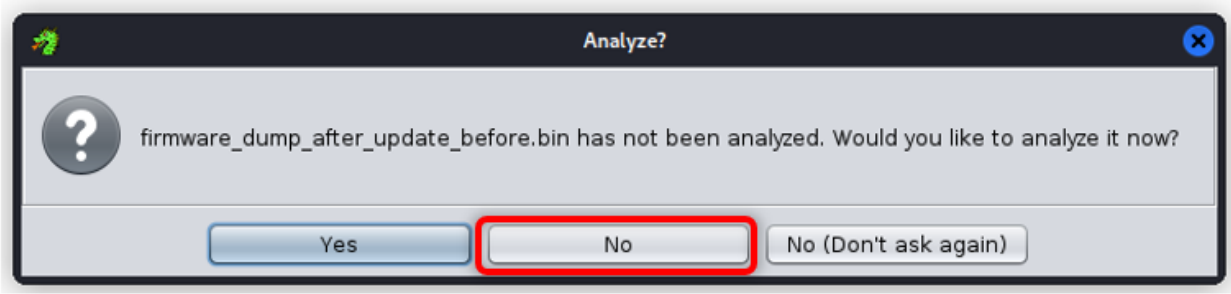
2.1 Chargement dans Ghidra

Nous commençons par charger le fichier du *firmware* dans le logiciel Ghidra en faisant glisser le fichier dedans. Après cela, la fenêtre suivante s'affiche. Nous définissons ensuite “ARM-Cortex-32-little" comme langage.



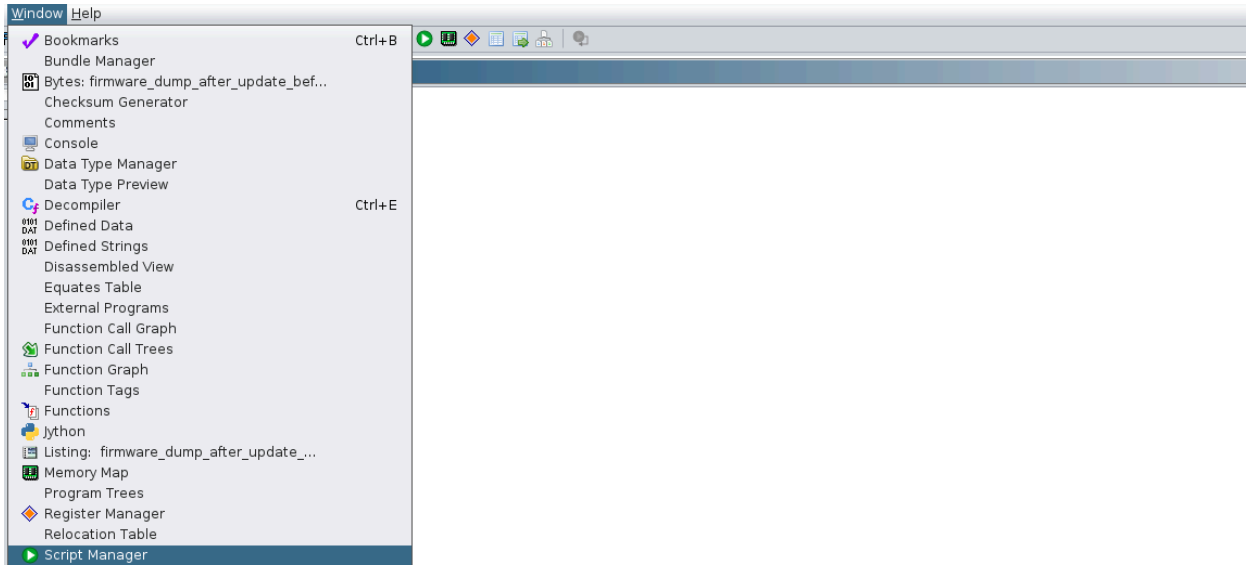


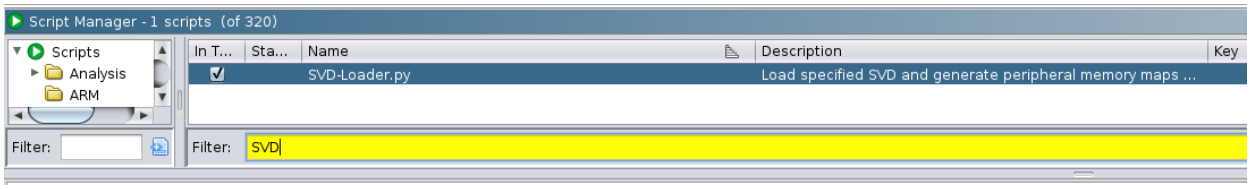
Nous double-cliquons maintenant sur le nom du fichier mais choisissons de ne pas l'analyser pour l'instant.

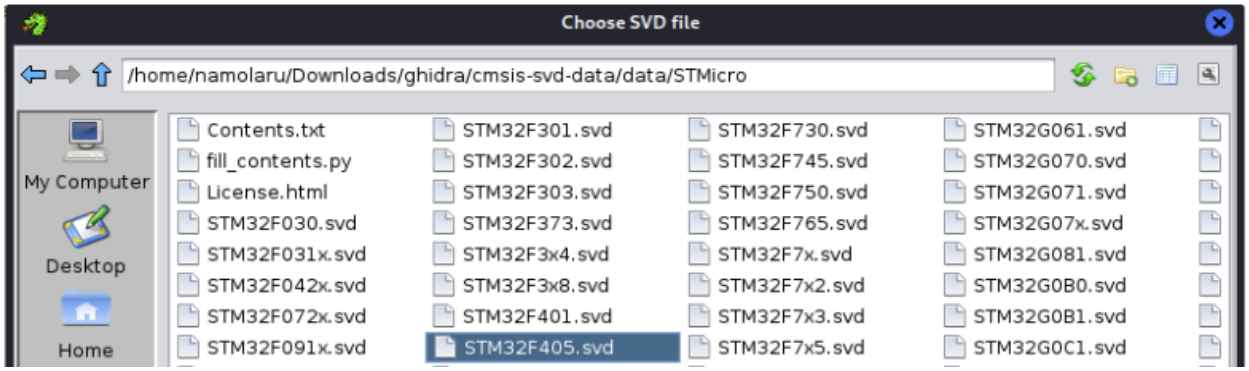


2.2 Chargement du SVD (System View Description)

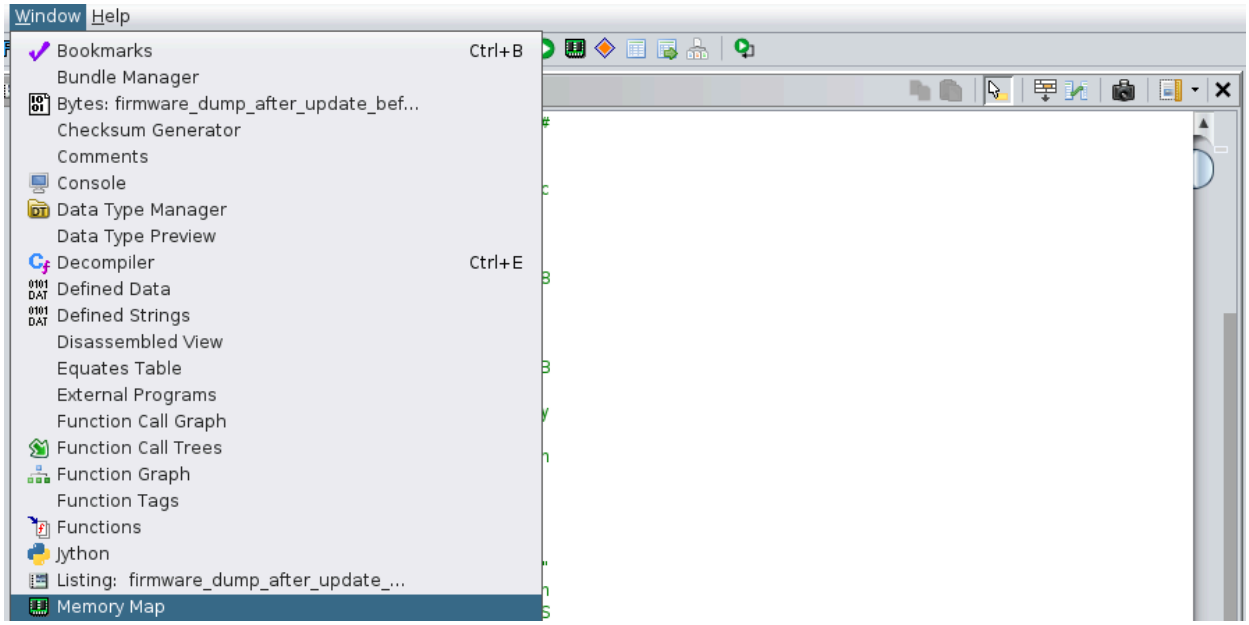
La première étape consiste à charger le fichier SVD correspondant.

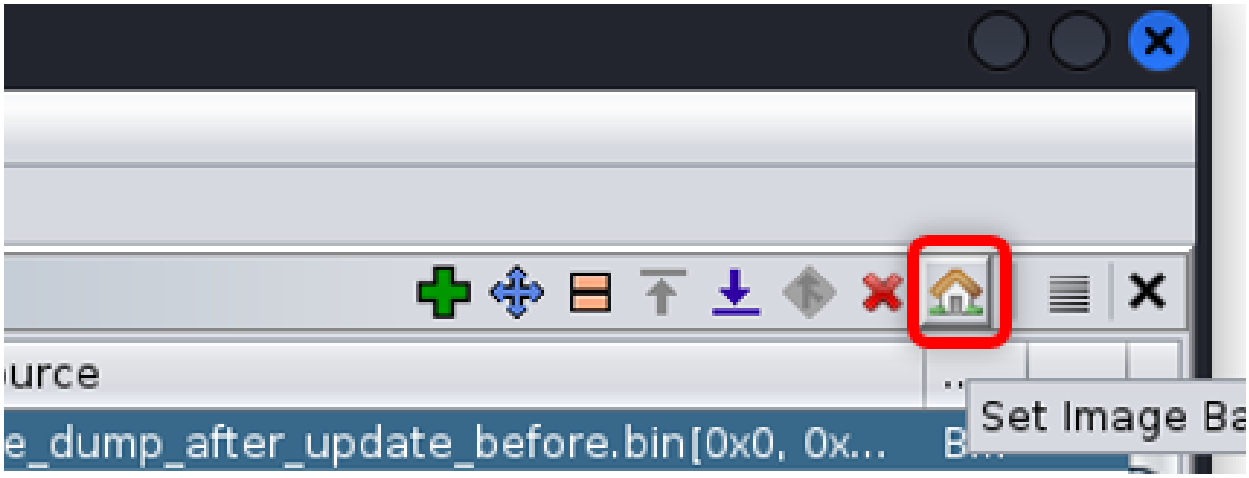


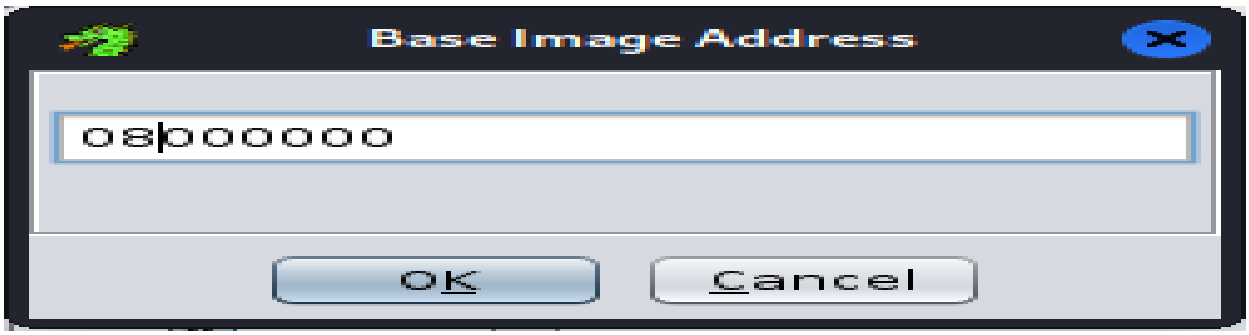




Ensuite, nous procédons à modifier l'adresse de base comme suit :

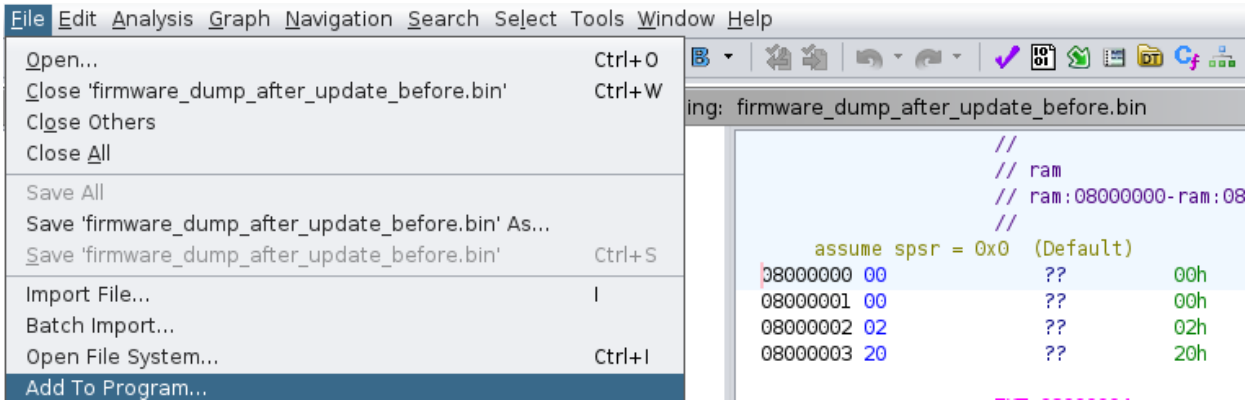

Window Help
Bookmarks Ctrl+B
Bundle Manager
Bytes: firmware_dump_after_update_bef...
Checksum Generator
Comments
Console
Data Type Manager
Data Type Preview
Decompiler Ctrl+E
Defined Data
Defined Strings
Disassembled View
Equates Table
External Programs
Function Call Graph
Function Call Trees
Function Graph
Function Tags
Functions
Jython
Listing: firmware_dump_after_update_...
Memory Map

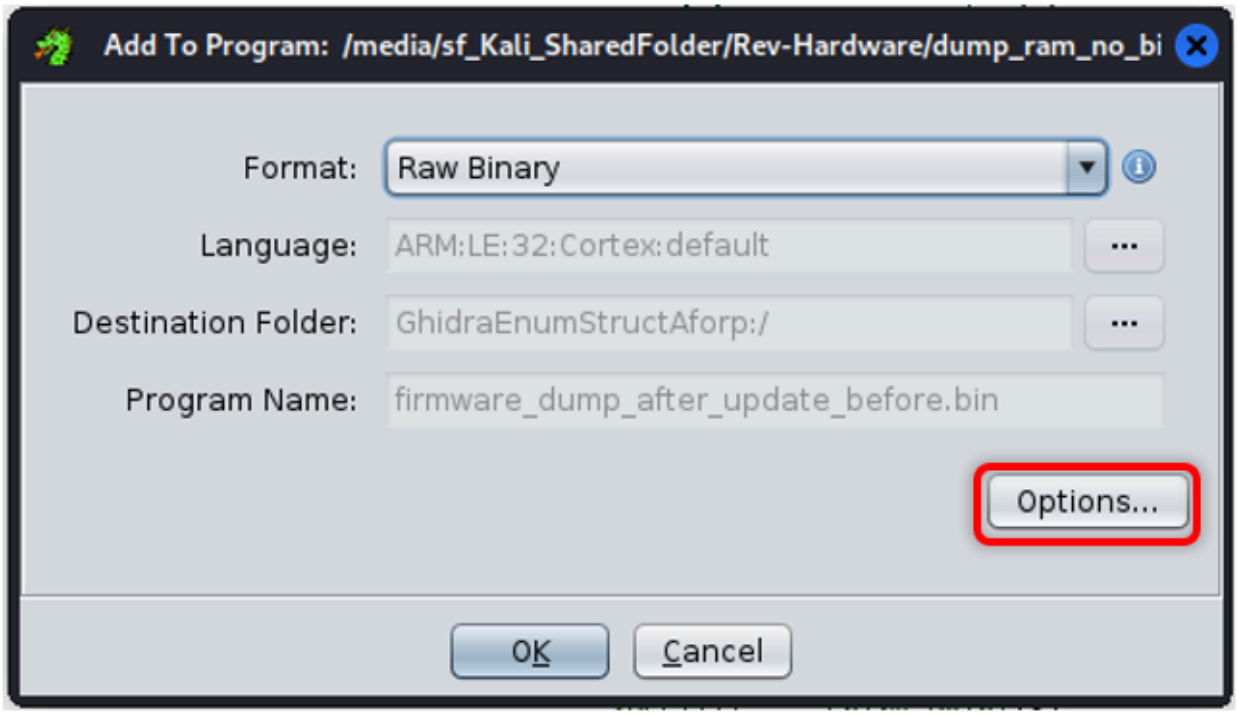

urce
e_dump_after_update_before.bin[0x0, 0x...
Set Image Ba

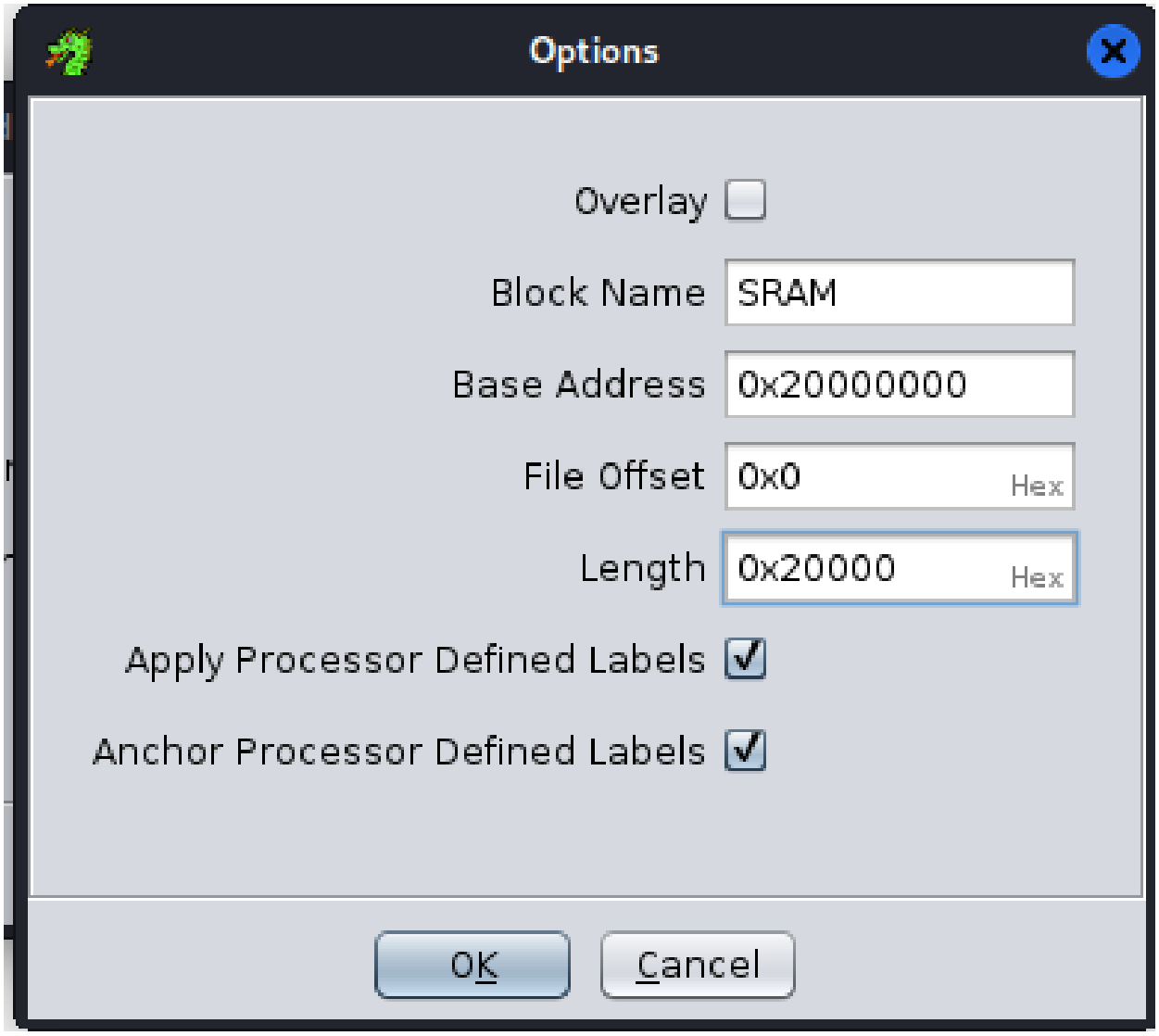

Base Image Address
08000000
OK
Cancel

2.3 Chargement de la SRAM

Une étape importante consiste à charger le *dump* de la RAM dans le logiciel. Pour ce faire, nous avons importer le fichier .bin de la RAM :

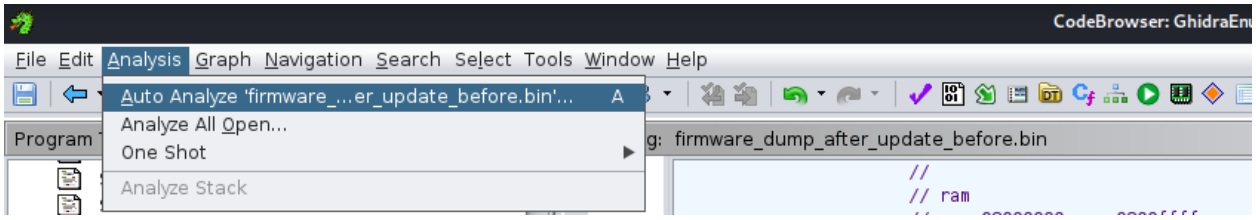


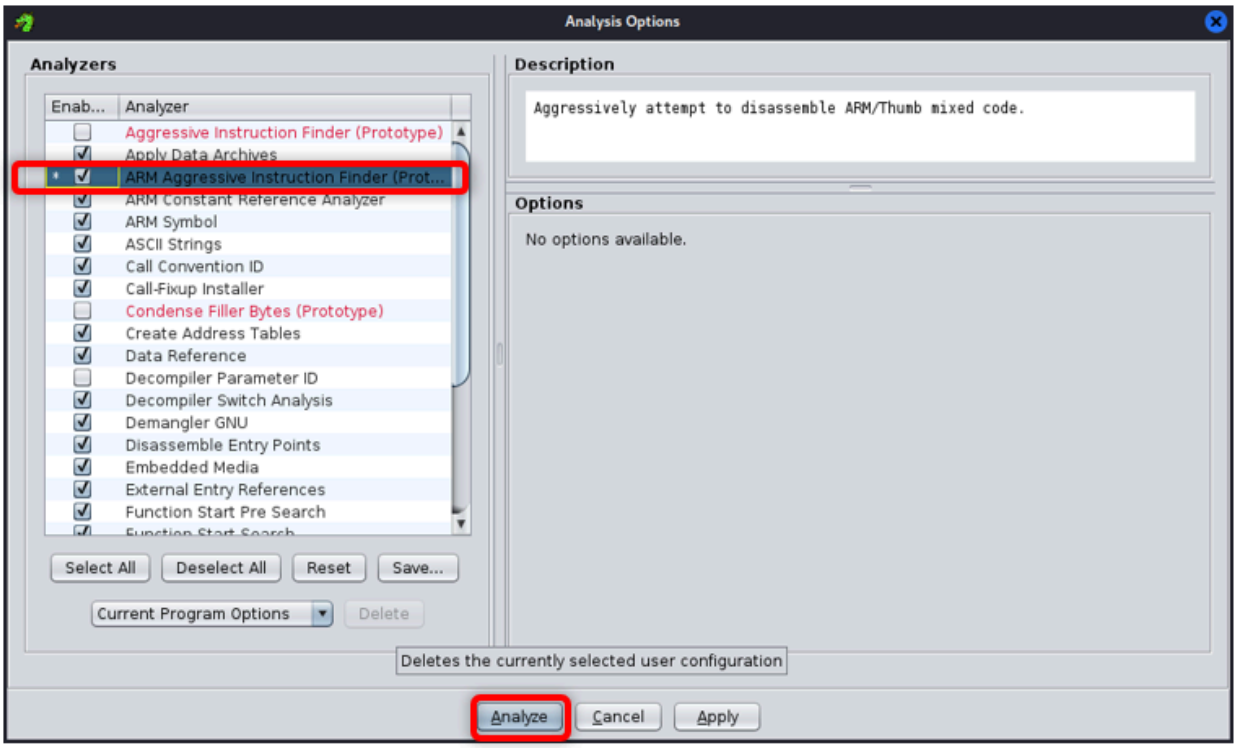




2.4 Analyse

Après avoir effectué toutes ces actions, nous avons lancé notre analyse.




Analysis Options
Analyzers
Enab...
Analyzer
Aggressive Instruction Finder (Prototype)
Apply Data Archives
ARM Aggressive Instruction Finder (Prot...
ARM Constant Reference Analyzer
ARM Symbol
ASCII Strings
Call Convention ID
Call-Fixup Installer
Condense Filler Bytes (Prototype)
Create Address Tables
Data Reference
Decompiler Parameter ID
Decompiler Switch Analysis
Demangler GNU
Disassemble Entry Points
Embedded Media
External Entry References
Function Start Pre Search
Select All
Deselect All
Reset
Save...
Current Program Options
Delete
Deletes the currently selected user configuration
Description
Aggressively attempt to disassemble ARM/Thumb mixed code.
Options
No options available.
Analyze
Cancel
Apply

3. Étude de l'authentification

3.1 Introduction

Il existe deux principales méthodes pour effectuer une analyse : la méthode « Haut en bas », qui consiste à démarrer l'analyse à partir du point d'entrée, et la méthode « Bas en haut », où nous partons des éléments qui attirent notre attention, tels que des chaînes de caractères ou des importations. Ainsi, nous commençons par rechercher la fonction qui utilise la chaîne de caractères `Enter_password:` afin de mieux comprendre le fonctionnement du mécanisme d'authentification.

De cette façon, nous trouvons la fonction `FUN_08000aa8` et nous commençons à l'analyser.

Decompile: FUN_08000aa8 - (firmware_dump_after_update_before.bin)

```
1
2  void FUN_08000aa8(void)
3
4  {
5    byte bVar1;
6    int iVar2;
7    undefined auStack_68 [32];
8    undefined auStack_48 [64];
9
10   while( true ) {
11     if (4 < *DAT_08000b28) {
12       FUN_0800066c(PTR_s_Too_many_failed_attempts._Access_08000b40);
13       return;
14     }
15     FUN_0800066c(PTR_s_Enter_password:_08000b30);
16     FUN_08000a20(auStack_48,0x40);
17     FUN_080007c4(auStack_68);
18     iVar2 = FUN_08000830(auStack_48,auStack_68);
19     if (iVar2 == 0) break;
20     bVar1 = *DAT_08000b28;
21     if ((1 < bVar1) && (iVar2 = FUN_080001d0(auStack_48,PTR_s_DEBUG123_08000b34), iVar2 == 0)) {
22       *DAT_08000b20 = 1;
23       *DAT_08000b38 = 1;
24       FUN_0800066c(PTR_s_[DEBUG_MODE_ENABLED]_Access_gran_08000b3c);
25       return;
26     }
27     *DAT_08000b28 = bVar1 + 1;
28     FUN_0800066c(DAT_08000b2c);
29   }
30   *DAT_08000b20 = 1;
31   FUN_0800066c(DAT_08000b24);
32   return;
33 }
34
```

À partir de cette section, nous concluons que la fonction FUN_0800066c est une fonction d'affichage. En programmation des systèmes embarqués, il est courant d'éviter d'utiliser printf, nous avons donc appeler cette fonction simplement print. Nous concluons également que DAT_08000b28 est un pointeur vers une zone mémoire qui stocke le nombre de tentatives de connexion incorrectes effectuées jusqu'à présent.

```
if (4 < *DAT_08000b28) {
  FUN_0800066c(PTR_s_Too_many_failed_attempts._Access_08000b40);
  return;
}
FUN_0800066c(PTR_s_Enter_password:_08000b30);
```

Étant donné que la variable **bVar1** stocke la valeur vers laquelle **DAT_08000b28** pointe, nous décidons d'appeler cette variable **attempts_cpt**.

```
bVar1 = *DAT_08000b28;
```

Les fonctions FUN_080001d0 et FUN_08000830 semblent être des fonctions de comparaison de chaînes de caractères, renvoyant zéro si les deux chaînes sont identiques. Alors que la fonction FUN_080001d0 ressemble à une fonction strcmp classique, la deuxième fonction ressemble à une fonction de comparaison plus personnalisée qui diffère des implémentations classiques. Dans les deux cas, la chaîne comparée est celle stockée dans auStack_48, ce qui permet de conclure que ce tampon contient l'entrée de l'utilisateur.

Le tampon auStack_48 est rempli à l'aide de la fonction FUN_080007c4 qui charge un contenu depuis la SRAM dans ce tampon (l'analyse de cette fonction fait l'objet de la section suivante.). Il s'agit du mot de passe que l'utilisateur doit saisir, et il est donc clair que la fonction FUN_08000830 compare le mot de passe saisi par l'utilisateur avec le vrai mot de passe.

Étant donné ces éléments, nous obtenons le code suivant.

```
void FUN_08000aa8(void)

{
  int cmpt_result;
  undefined real_password_buffer [32];
  undefined user_buffer [64];
  byte attempts_cpt;

  while( true ) {
    if (4 < *attempts_cpt_ptr) {
      print(PTR_s_Too_many_failed_attempts._Access_08000b40);
      return;
    }
    print(PTR_s_Enter_password:_08000b30);
    FUN_08000a20(user_buffer,0x40);
    FUN_080007c4(real_password_buffer);
    cmpt_result = FUN_08000830(user_buffer,real_password_buffer);
    if (cmpt_result == 0) break;
    attempts_cpt = *attempts_cpt_ptr;
    if ((1 < attempts_cpt) &&
       (cmpt_result = strcmp(user_buffer,PTR_s_DEBUG123_08000b34), cmpt_result == 0)) {
      *DAT_08000b20 = 1;
      *DAT_08000b38 = 1;
      print(PTR_s_[DEBUG_MODE_ENABLED]_Access_gran_08000b3c);
      return;
    }
    *attempts_cpt_ptr = attempts_cpt + 1;
    print(DAT_08000b2c);
  }
  *DAT_08000b20 = 1;
  print(DAT_08000b24);
  return;
}
```

3.2 Stockage du mot de passe

Comme déjà indiqué dans la section précédente, le rôle de la fonction **FUN_080007c4** est de charger le vrai mot de passe dans un tampon. Nous découvrons ainsi que ce mot de passe est basé sur une liste de caractères stockés dans la SRAM. Afin d'obtenir le mot de passe, une opération XOR est effectuée entre la valeur hexadécimale de chacun de ces caractères et la valeur **0x5a**. De cette manière, le mot de passe est reconstitué et utilisé pour la comparaison avec celui saisi par l'utilisateur. De plus, nous pouvons voir que, sous certaines conditions, ce mot de passe est affiché à l'utilisateur.

...

```
void FUN_080007c4(int buffer_ptr)

{
  uint i;

  for (i = 0; i < 7; i = i + 1) {
    *(byte *)(buffer_ptr + i) = *(byte *)(DAT_08000800 + i) ^ 0x5a;
  }
  *(undefined *)(buffer_ptr + 7) = 0;
  if (*DAT_08000804 == '\x01') {
    print([DEBUG]_Decrypted_password);
    print(buffer_ptr);
    print(Entering_TLV_command_mode);
  }
  return;
}
```

```
                    DAT_08000800                            XREF[1]:     FUN_080007c4:080007cc(R)
08000800 14 00 00 20     undefined4 20000014h                              ; ?  ->  20000014

                    s_6?.7?34_20000014                      XREF[1]:     FUN_080007c4:080007ce(R)
20000014 36 3f 2e        ds         "6?.7?34"
         37 3f 33
         34 00
```

buffer = [0x36, 0x3f, 0x2e, 0x37, 0x3f, 0x33, 0x34]
password = ''

```
for i in range(0, 7):
        password += chr(buffer[i] ^ 0x5a)

print(password)
```

```
┌──(hardware-venv)─(namolaru㉿kali)-[/media/sf_Kali_SharedFolder/Rev-Hardware]
└─$ python main.py
letmein
```

Mais que représente **DAT_08000804** ?

```
                    DAT_08000804                                    XREF[1]:     FUN_080007c4:080007e0(R)
08000804 95 01 00 20     undefined4  20000195h                                    ; ?  ->  20000195
```

Il s'agit d'un pointeur vers une zone mémoire située dans la SRAM, référencée par plusieurs autres fonctions.

```
                    DAT_20000195                                    XREF[6]:     FUN_08000680:08000686(R),
                                                                                 FUN_08000734:0800073a(R),
                                                                                 FUN_0800078c:08000790(R),
                                                                                 FUN_080007c4:080007e2(R),
                                                                                 FUN_08000aa8:08000b0a(*),
                                                                                 FUN_08000aa8:08000b0c(W)
20000195 00              undefined1  00h
```

Par exemple, dans la fonction **FUN_08000680**, le pointeur vers cette zone de mémoire s'appelle **DAT_08000710**, et dans le cas où la valeur stockée sur la SRAM est nulle, un message s'affiche à l'utilisateur (avec la fonction d'affichage **FUN_0800066c**) :

```
if (*DAT_08000710 == '\0') {
  FUN_0800066c(DAT_08000720);
}
```

Nous commençons par regarder quel message est affiché à l'utilisateur.

```
                    DAT_08000720                                    XREF[1]:     FUN_08000680:080006a0(R
08000720 94 4c 00 08     undefined4 08004C94h                                        ; ?  ->  08004c94

                    s_[-]_Debug_mode_required._08004c94             XREF[0,3]:   FUN_08000680:080006a0(*
                                                                                 FUN_08000734:08000774(*
                                                                                 FUN_0800078c:080007ac(*
08004c91 46 70 47        ds         "FpG[-] Debug mode required.\r\n"
         5b 2d 5d
         20 44 65 ...
```

Et nous en concluons que le but de la condition est de vérifier si le mode **DEBUG** est déjà activé. Nous obtenons alors :

```
if (*debugModeEnabled == '\0') {
  print(DebugModeRequired);
}
```

Et pour la fonction **FUN_080007c4**, nous concluons que si le mode **DEBUG** est activé, le vrai mot de passe est affiché à l'utilisateur après avoir été chargé dans la mémoire tampon.

```
if (*DebugModEnabled == '\x01') {
  print([DEBUG]_Decrypted_password);
  print(buffer_ptr);
  print(Entering_TLV_command_mode);
}
```

3.3 Activation du mode debug

L'examen de la fonction **FUN_08000aa8** nous apprend également comment activer le mode **DEBUG** : pour cela, l'utilisateur doit effectuer au moins une tentative de connexion incorrecte, puis saisir **DEBUG123** comme mot de passe. Reste maintenant à comprendre la signification de **DAT_08000b20** et **DAT_08000b38**.

```
if ((1 < attempts_cpt) &&
   (cmp_result = strcmp(user_buffer,PTR_s_DEBUG123_08000b34), cmp_result == 0)) {
  *DAT_08000b20 = 1;
  *DAT_08000b38 = 1;
  print(PTR_s_[DEBUG_MODE_ENABLED]_Access_gran_08000b3c);
  return;
}
```

Pour **DAT_08000b20** ,

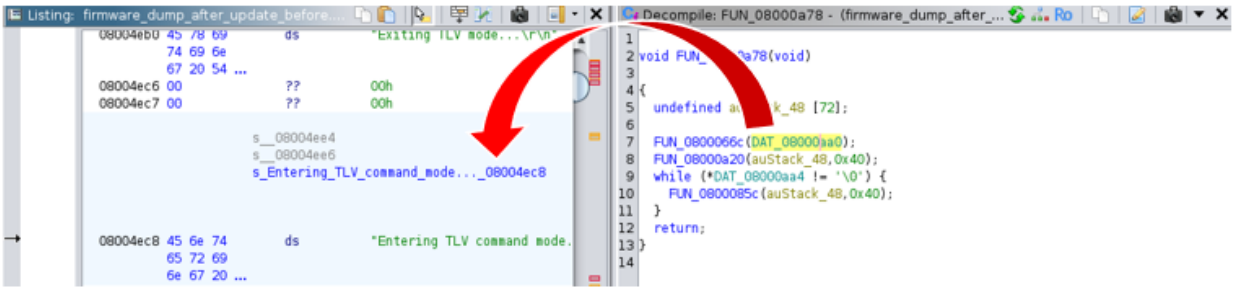
```
                    DAT_08000b20                                XREF[2]:     FUN_08000aa8:08000aae(R),
                                                                             FUN_08000aa8:08000b06(R)
08000b20 94 01 00 20     undefined4  20000194h                                ; ?  ->  20000194
```

```
                    DAT_20000194                                XREF[9]:     FUN_0800085c:08000892(*),
                                                                             FUN_0800085c:08000896(W),
                                                                             FUN_0800085c:080008e4(*),
                                                                             FUN_0800085c:080008e8(W),
                                                                             FUN_08000a78:08000a8c(R),
                                                                             FUN_08000aa8:08000aae(*),
                                                                             FUN_08000aa8:08000ab2(W),
                                                                             FUN_08000aa8:08000b08(W),
                                                                             FUN_08000b44:08000b60(R)
20000194 00              undefined1  00h
```

Nous en concluons que **DAT_08000b70** de la fonction **FUN_08000b44** pointe également vers la même zone mémoire. Si la valeur stockée dans cette zone mémoire est égale à zéro, la fonction **FUN_08000aa8** est appelée, sinon la fonction **FUN_08000a78** est appelée.

```
if (*DAT_08000b70 == '\0') {
  FUN_08000aa8();
}
else {
  FUN_08000a78();
}
```

FUN_08000aa8 est la fonction par laquelle nous avons démarré l'analyse et elle gère tout ce qui concerne l'authentification. La fonction **FUN_08000a78** s'occupe de la gestion du mode TLV :



Le but de la condition ci-dessous est donc de vérifier si le mode TLV est activé :

```
if (*TlvModeEnabled == '\0') {
  FUN_08000aa8();
}
else {
  FUN_08000a78();
}
```

Pour **DAT_08000b38** ,

```
                    DAT_08000b38                                    XREF[1]:     FUN_08000aa8:08000b0a(R)
08000b38 95 01 00 20     undefined4  20000195h                                    ; ?  ->  20000195
```

```
                    DAT_20000195                                    XREF[6]:     FUN_08000680:08000686(R),
                                                                                 FUN_08000734:0800073a(R),
                                                                                 FUN_0800078c:08000790(R),
                                                                                 FUN_080007c4:080007e2(R),
                                                                                 FUN_08000aa8:08000b0a(*),
                                                                                 FUN_08000aa8:08000b0c(W)
20000195 00              undefined1  00h
```

Nous avons déjà identifié qu'il s'agit de la zone mémoire qui indique si le mode **DEBUG** est activé.

Sur la base de ces données, nous pouvons désormais mieux comprendre le code.

```
void FUN_08000aa8(void)

{
  int cmpt_result;
  undefined real_password_buffer [32];
  undefined user_buffer [64];
  byte attempts_cpt;

  while( true ) {
    if (4 < *attempts_cpt_ptr) {
      print(PTR_s_Too_many_failed_attempts._Access_08000b40);
      return;
    }
    print(PTR_s_Enter_password:_08000b30);
    FUN_08000a20(user_buffer,0x40);
    FUN_080007c4(real_password_buffer);
    cmpt_result = FUN_08000830(user_buffer,real_password_buffer);
    if (cmpt_result == 0) break;
    attempts_cpt = *attempts_cpt_ptr;
    if ((1 < attempts_cpt) &&
       (cmpt_result = strcmp(user_buffer,PTR_s_DEBUG123_08000b34), cmpt_result == 0)) {
      *TlvModeEnabled = 1;
      *debugModeEnabled = 1;
      print(PTR_s_[DEBUG_MODE_ENABLED]_Access_gran_08000b3c);
      return;
    }
    *attempts_cpt_ptr = attempts_cpt + 1;
    print(DAT_08000b2c);
  }
  *TlvModeEnabled = 1;
  print(DAT_08000b24);
  return;
}
```

3.4 Fonction de validation du mot de passe

La fonction de comparaison vérifie, caractère par caractère, si le mot de passe saisi par l'utilisateur correspond au mot de passe réel. La vérification s'arrête dès qu'un caractère non identique est détecté ou lorsque la fin de l'une des chaînes est atteinte. Après chaque correspondance de caractères, un appel à la fonction **FUN_08002d44** est effectué.

```
int FUN_08000830(int buffer_addr_1,int buffer_addr_2)

{
  uint current_char;
  int i;

  for (i = 0; (current_char = (uint)*(byte *)(buffer_addr_1 + i), current_char != 0 &&
              (*(byte *)(buffer_addr_2 + i) != 0)); i = i + 1) {
    if (current_char != *(byte *)(buffer_addr_2 + i)) {
      return 1;
    }
    FUN_08002d44(0x32);
  }
  i = current_char - *(byte *)(buffer_addr_2 + i);
  if (i != 0) {
    i = 1;
  }
  return i;
}
```

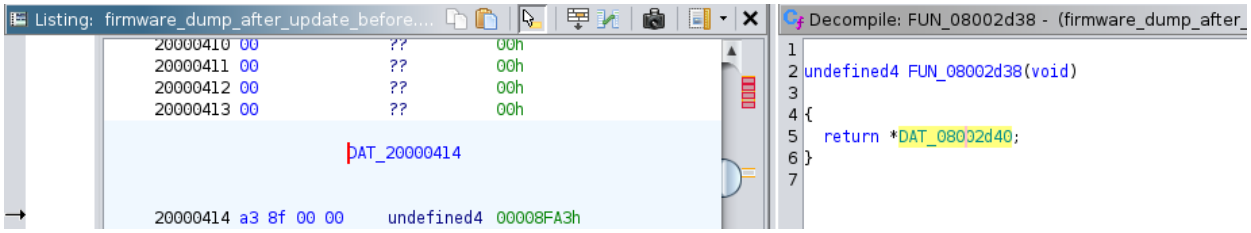
La fonction **FUN_08002d44** semble effectuer des itérations de boucles infinies :

```
void FUN_08002d44(uint param_1)

{
  int iVar1;
  int iVar2;

  iVar1 = FUN_08002d38();
  if (param_1 != 0xffffffff) {
    param_1 = param_1 + *DAT_08002d68;
  }
  do {
    iVar2 = FUN_08002d38();
  } while ((uint)(iVar2 - iVar1) < param_1);
  return;
}
```

Nous pourrions par exemple effectuer les mêmes opérations en utilisant le code Python suivant, car la fonction **FUN_08002d38** renvoie la valeur stockée dans la zone mémoire suivante :



De plus, DAT_08002d68 pointe vers cette zone de mémoire :

```
                    DAT_2000001c                    XREF[3]:   FUN_08002ca0:08002ca6(R),
                                                               08002d34(*),
                                                               FUN_08002d44:08002d56(R)
2000001c 01         undefined1  01h
2000001d 00         ??          00h
2000001e 00         ??          00h
2000001f 00         ??          00h
```

```
def FUN_08002d38():
    return 0xa38f0000


def FUN_08002d44(param_1):
    iVar1 = FUN_08002d38()
    if (param_1 != 0xffffffff):
        param_1 + param_1 + 0x01000000


    iVar2 = FUN_08002d38()
    while (iVar2 - iVar1) < param_1:
        print((iVar2 - iVar1), param_1)
        iVar2 = FUN_08002d38()


FUN_08002d44(0x32)
```

Cependant, il s’agit de zones de mémoire accessibles par d’autres fonctions. Une estimation raisonnable est donc que les valeurs dans ces zones de mémoire changent en même temps, de sorte que le nombre d'itérations de boucle sera fini. Cela implique qu'il y a un certain délai après la vérification de chaque caractère du mot de passe, rendant cette fonction vulnérable aux attaques temporelles (*Timing attack*).

4. Analyse des commandes TLV

4.1 Le schéma de codage TLV dans le code C

TLV (type-length-value ou tag-length-value) est un schéma de codage utilisé pour les éléments d'information dans les protocoles de communication. Un flux de données codé en TLV contient du code relatif au type d'enregistrement, à la longueur de la valeur de l'enregistrement et enfin à la valeur elle-même.[1] Nous avons vu précédemment que la fonction FUN_08000a78 est responsable de la gestion du mode TLV.

```
void FUN_08000a78(void)

{
  undefined auStack_48 [72];

  FUN_0800066c(Entering_TLV_command_mode);
  FUN_08000a20(auStack_48,0x40);
  while (*DAT_08000aa4 != '\0') {
    FUN_0800085c(auStack_48,0x40);
  }
  return;
}
```

DAT_08000aa4 pointe vers la même zone en mémoire que le pointeur dans la condition de la fonction FUN_08000b44 :

```
if (*TlvModeEnabled == '\0') {
  FUN_08000aa8();
}
else {
  FUN_08000a78();
}
```

[1] https://en.wikipedia.org/wiki/Type%E2%80%93length%E2%80%93value

Nous en concluons qu'il s'agit de la même condition et que **auStack_48** est le tampon pour le TLV que l'utilisateur va saisir.

```
void FUN_08000a78(void)

{
  undefined tlv_buffer [72];

  print(Entering_TLV_command_mode);
  FUN_08000a20(tlv_buffer,0x40);
  while (*DAT_08000aa4 != '\0') {
    FUN_0800085c(tlv_buffer,0x40);
  }
  return;
}
```

Alors qu'il semble que le rôle de la fonction **FUN_08000a20** soit de recevoir les entrées de l'utilisateur, le rôle de la fonction **FUN_0800085c** est de les interpréter. Pour comprendre la signification du code, nous commençons par trouver les chaînes de caractères correspondantes.

```
if (param_2 < 3) {
  FUN_0800066c(Invalid_TLV_command);
}
else {
  bVar1 = *param_1;
  uVar2 = *(undefined2 *)(param_1 + 1);
  FUN_08004554(auStack_50,[DEBUG]_TLV_Received_-_Type:_0x%,bVar1,uVar2);
  FUN_0800066c(auStack_50);
  if (bVar1 == 0xaa) {
    FUN_08000680();
  }
  else {
    if (bVar1 < 0xab) {
      if (bVar1 == 3) {
        FUN_080043a4(*DAT_08000958);
        FUN_0800066c([+]_Memory_freed);
        return;
      }
      if (bVar1 < 4) {
        if (bVar1 == 1) {
          *DAT_08000964 = uVar2;
          iVar4 = FUN_08004394(uVar2);
          *DAT_08000958 = iVar4;
          if (iVar4 != 0) {
            FUN_0800066c([+]_Memory_allocated);
            return;
          }
          FUN_0800066c([-]_Allocation_failed);
          return;
        }
        if (bVar1 == 2) {
          if (*DAT_08000958 != 0) {
            FUN_080043b4(*DAT_08000958,param_1 + 3,uVar2);
            FUN_0800066c([+]_Heap_overflow_triggered);
            return;
          }
          FUN_0800066c([-]_No_allocated_memory);
          return;
        }
      }
```

param_1 est le tampon, param_2 est sa longueur et nous comprenons que bVar1 est la partie Type du TLV tandis que uVar2 est la partie Length. FUN_0800066c est, comme nous l'avons vu précédemment, la fonction d'affichage.

```
void FUN_0800085c(byte *tlv_buffer,uint tlv_buffer_len)

{
  undefined4 uVar1;
  int iVar2;
  undefined auStack_70 [32];
  undefined auStack_50 [64];
  undefined2 tlv_length;
  byte tlv_type;

  if (tlv_buffer_len < 3) {
    print(Invalid_TLV_command);
  }
  else {
    tlv_type = *tlv_buffer;
    tlv_length = *(undefined2 *)(tlv_buffer + 1);
    FUN_08004554(auStack_50,[DEBUG]_TLV_Received_-_Type:_0x%,tlv_type,tlv_length);
    print(auStack_50);
```

Cela nous permet de conclure que la partie Type du TLV est codée sur un octet, et qu'immédiatement après se trouve la partie Length. Nous identifions également les valeurs suivantes pour les types de commandes possibles : 3, 1, 2, 0x42, 0xcc, 0xff, et 0xbb.

4.2 Gestion des allocations mémoire

- Type 3 : Libération de l'allocation mémoire pointée par *DAT_08000958.

```
if (tlv_type == 3) {
  FUN_080043a4(*DAT_08000958);
  print([+]_Memory_freed);
  return;
}
```

La fonction **FUN_080043a4** est donc très probablement la fonction **free**.

```
    if (tlv_type == 3) {
      free(*ptr_to_malloc_result);
      print([+]_Memory_freed);
      return;
    }
```

- Type **1** : Demande d'allocation mémoire de taille **tlv_length**.

```
if (tlv_type == 1) {
  *DAT_08000964 = tlv_length;
  iVar2 = FUN_08004394(tlv_length);
  *ptr_to_malloc_result = iVar2;
  if (iVar2 != 0) {
    print([+]_Memory_allocated);
    return;
  }
  print([-]_Allocation_failed);
  return;
}
```

La fonction **FUN_08004394** est donc très probablement la fonction **malloc**.

```
if (tlv_type == 1) {
  *ptr_to_malloc_length = tlv_length;
  malloc_result = malloc(tlv_length);
  *ptr_to_malloc_result = malloc_result;
  if (malloc_result != 0) {
    print([+]_Memory_allocated);
    return;
  }
  print([-]_Allocation_failed);
  return;
}
```

- Type 2 : D'après la condition qui apparaît, nous comprenons que pour utiliser cette commande, une zone mémoire doit d'abord être allouée à l'aide de la commande 1.

```
if (tlv_type == 2) {
  if (*ptr_to_malloc_result != 0) {
    FUN_080043b4(*ptr_to_malloc_result,tlv_buffer + 3,tlv_length);
    print([+]_Heap_overflow_triggered);
    return;
  }
  print([-]_No_allocated_memory);
  return;
}
```

Mais, pour mieux comprendre cette partie, il faut d'abord s'intéresser à la fonction FUN_080043b4.

```
void FUN_080043b4(int param_1,undefined *param_2,int param_3)

{
  undefined *puVar1;
  undefined *puVar2;
  undefined *puVar3;

  puVar2 = param_2 + param_3;
  puVar3 = (undefined *)(param_1 + -1);
  if (param_2 != puVar2) {
    do {
      puVar1 = param_2 + 1;
      puVar3 = puVar3 + 1;
      *puVar3 = *param_2;
      param_2 = puVar1;
    } while (puVar1 != puVar2);
    return;
  }
  return;
}
```

En donnant de nouveaux noms aux paramètres, tout devient plus clair :

```
void FUN_080043b4(int malloc_result_addr,undefined *buffer,int buffer_len)

{
  undefined *puVar1;
  undefined *puVar2;
  undefined *puVar3;

  puVar2 = buffer + buffer_len;
  puVar3 = (undefined *)(malloc_result_addr + -1);
  if (buffer != puVar2) {
    do {
      puVar1 = buffer + 1;
      puVar3 = puVar3 + 1;
      *puVar3 = *buffer;
      buffer = puVar1;
    } while (puVar1 != puVar2);
    return;
  }
  return;
}
```

Nous en concluons que **puVar2** est un pointeur vers la fin du tampon, **puVar3** pointe vers l'emplacement suivant dans le tampon alloué à l'aide de **malloc**, et que **puVar1** pointe vers le caractère suivant dans le tampon **buffer**. Ce qui nous donne :

```
void FUN_080043b4(int malloc_result_addr,undefined *buffer,int buffer_len)

{
  undefined *next_char_in_buffer;
  undefined *ptr_buffer_end;
  undefined *malloc_buffer_ptr;

  ptr_buffer_end = buffer + buffer_len;
  malloc_buffer_ptr = (undefined *)(malloc_result_addr + -1);
  if (buffer != ptr_buffer_end) {
    do {
      next_char_in_buffer = buffer + 1;
      malloc_buffer_ptr = malloc_buffer_ptr + 1;
      *malloc_buffer_ptr = *buffer;
      buffer = next_char_in_buffer;
    } while (next_char_in_buffer != ptr_buffer_end);
    return;
  }
  return;
}
```

Nous comprenons de cela que le but de la fonction **FUN_080043b4** est de copier la partie **Value** de la commande **TLV** dans un tampon alloué par **malloc**, et puisque la copie commence à partir de **tlv_buffer + 3**, nous comprenons que la partie **Length** de la commande **TLV** est codée sur deux octets.

```
if (tlv_type == 2) {
  if (*ptr_to_malloc_result != 0) {
    buffer_copy(*ptr_to_malloc_result,tlv_buffer + 3,tlv_length);
    print([+]_Heap_overflow_triggered);
    return;
  }
  print([-]_No_allocated_memory);
  return;
}
```

4.3 Faire crasher le système et dump des registres

- Type **0xcc** : Faire crasher le système

```
  if (tlv_type == 0xcc) {
    FUN_0800078c();
    return;
  }
```

La première étape consiste à examiner le contenu de la fonction **FUN_0800078c** :

```
void FUN_0800078c(void)

{
  if (*DAT_080007b4 != '\0') {
    print(DAT_080007b8);
    FUN_08000680();
    FUN_08000734();
    print(DAT_080007bc);
    disableIRQinterrupts();
    do {
                    /* WARNING: Do nothing block with infinite loop */
    } while( true );
  }
  print(DAT_080007c0);
  return;
}
```

Avec les chaînes de caractères :

```
void FUN_0800078c(void)

{
  if (*DAT_080007b4 != '\0') {
    print([!]_SYSTEM_CRASH_DETECTE);
    FUN_08000680();
    FUN_08000734();
    print([!]_Halting_system);
    disableIRQinterrupts();
    do {
                    /* WARNING: Do nothing block with infinite loop */
    } while( true );
  }
  print([-]_Debug_mode_required);
  return;
}
```

DAT_080007b4 pointe vers la même zone mémoire que celle que avons déjà identifiée comme la zone où est stockée la valeur indiquant si le mode DEBUG est activé ou non. Cette commande permet donc de provoquer un crash dans le système, mais elle n'est disponible qu'en mode DEBUG.

```
void FUN_0800078c(void)

{
  if (*DebugModeEnabled != '\0') {
    print([!]_SYSTEM_CRASH_DETECTE);
    FUN_08000680();
    FUN_08000734();
    print([!]_Halting_system);
    disableIRQinterrupts();
    do {
                    /* WARNING: Do nothing block with infinite loop */
    } while( true );
  }
  print([-]_Debug_mode_required);
  return;
}
```

- Type 0xbb : dump des registres

```
if (tlv_type == 0xbb) {
  FUN_08000734();
  return;
}
```

La première étape consiste à examiner le contenu de la fonction **FUN_08000734** :

```
void FUN_08000734(void)

{
  bool bVar1;
  uint uVar2;
  uint uVar3;
  undefined4 in_r12;
  undefined4 uVar4;
  char in_NG;
  char in_ZR;
  char in_CY;
  char in_OV;
  byte in_Q;
  undefined8 uVar5;
  undefined auStack_118 [260];

  if (*DAT_0800077c == '\0') {
    print(DAT_08000788);
  }
  else {
    uVar4 = 0x8000745;
    uVar5 = print(DAT_08000780);
    uVar3 = (uint)(byte)(in_NG << 4 | in_ZR << 3 | in_CY << 2 | in_OV << 1 | in_Q) << 0x1b;
    bVar1 = (bool)isCurrentModePrivileged();
    if (bVar1) {
      uVar2 = getCurrentExceptionNumber();
      uVar3 = uVar3 | uVar2 & 0x1f;
    }
    FUN_08004554(auStack_118,DAT_08000784,(int)uVar5,(int)((ulonglong)uVar5 >> 0x20),uVar5,in_r12,
                 uVar4,0x8000754,uVar3);
    print(auStack_118);
  }
  return;
}
```

Avec les chaînes de caractères :

```
if (*DAT_0800077c == '\0') {
  print([-]_Debug_mode_required);
}
else {
  uVar4 = 0x8000745;
  uVar5 = print([DEBUG]_Register_Dump);
```

DAT_0800077c pointe vers la même zone mémoire que celle que avons déjà identifiée comme la zone où est stockée la valeur indiquant si le mode **DEBUG** est activé ou non. Cette commande permet donc de dump des registres mais elle n'est disponible qu'en mode DEBUG.

4.4 Types de commandes supplémentaires

- Type 0xff : Sortir du mode TLV

```
if (tlv_type == 0xff) {
  *DAT_0800094c = 0;
  print(Exiting_TLV_mode);
  return;
}
```

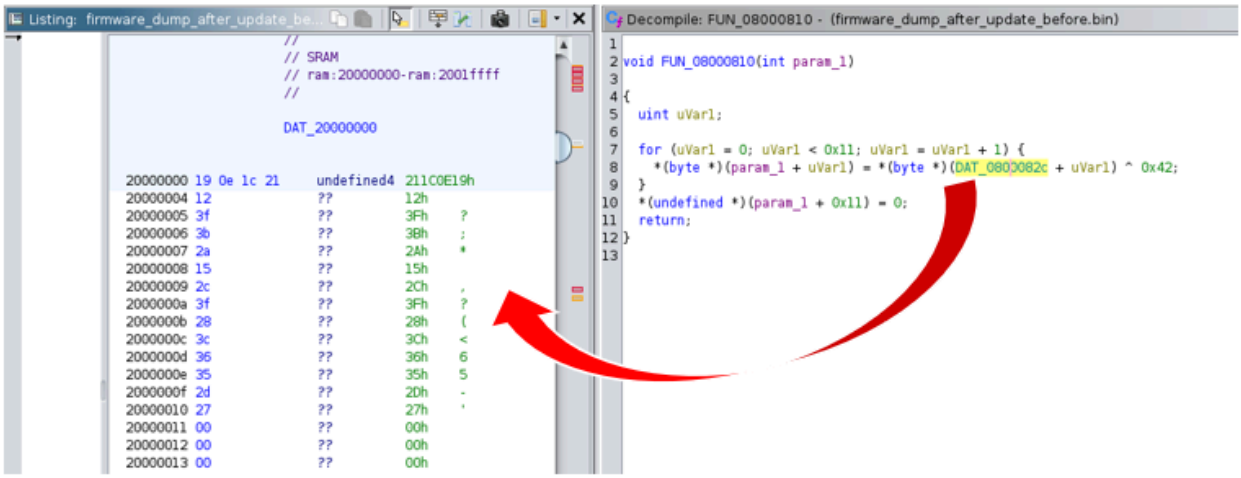
DAT_0800094c pointe vers la même zone mémoire que celle que nous avons déjà identifiée comme la zone où est stockée la valeur indiquant si le mode TLV est activé ou non.

```
if (tlv_type == 0xff) {
  *TlvModeEnabled = 0;
  print(Exiting_TLV_mode);
  return;
}
```

- Type 0x42 : Dump du flag et sortir du mode TLV

```
  else if (tlv_type == 0x42) {
    *TlvModeEnabled = 0;
    FUN_08000810(auStack_70);
    uVar1 = Entering_TLV_command_mode;
    print(Entering_TLV_command_mode);
    print(auStack_70);
    print(uVar1);
    return;
  }
}
```

La première étape consiste à examiner le contenu de la fonction FUN_08000810 : cette fonction charge du contenu depuis la SRAM dans le tampon auStack_70.



Comme cela sera expliqué dans les chapitres suivants du rapport, ce sont des éléments qui permettront de retrouver le Flag. Enfin, nous remarquons que cette commande, comme la commande **0xff**, désactive le mode **TLV**, ce qui est étrange, car cela contredit les messages affichés à l'utilisateur.

4.5 Récapitulatif

```
void FUN_0800085c(byte *tlv_buffer,uint tlv_buffer_len)

{
  undefined4 uVar1;
  int malloc_result;
  undefined auStack_70 [32];
  undefined auStack_50 [64];
  undefined2 tlv_length;
  byte tlv_type;

  if (tlv_buffer_len < 3) {
    print(Invalid_TLV_command);
  }
  else {
    tlv_type = *tlv_buffer;
    tlv_length = *(undefined2 *)(tlv_buffer + 1);
    FUN_08004554(auStack_50,[DEBUG]_TLV_Received_-_Type:_0x%,tlv_type,tlv_length);
    print(auStack_50);
    if (tlv_type == 0xaa) {
      FUN_08000680();
    }
    else {
      if (tlv_type < 0xab) {
        if (tlv_type == 3) {
          free(*ptr_to_malloc_result);
          print([+]_Memory_freed);
          return;
        }
        if (tlv_type < 4) {
          if (tlv_type == 1) {
            *ptr_to_malloc_length = tlv_length;
            malloc_result = malloc(tlv_length);
            *ptr_to_malloc_result = malloc_result;
            if (malloc_result != 0) {
              print([+]_Memory_allocated);
              return;
            }
            print([-]_Allocation_failed);
            return;
          }
```

```
        if (tlv_type == 2) {
          if (*ptr_to_malloc_result != 0) {
            buffer_copy(*ptr_to_malloc_result,tlv_buffer + 3,tlv_length);
            print([+]_Heap_overflow_triggered);
            return;
          }
          print([-]_No_allocated_memory);
          return;
        }
      }
      else if (tlv_type == 0x42) {
        *TlvModeEnabled = 0;
        FUN_08000810(auStack_70);
        uVar1 = Entering_TLV_command_mode;
        print(Entering_TLV_command_mode);
        print(auStack_70);
        print(uVar1);
        return;
      }
    }
    else {
      if (tlv_type == 0xcc) {
        FUN_0800078c();
        return;
      }
      if (tlv_type == 0xff) {
        *TlvModeEnabled = 0;
        print(Exiting_TLV_mode);
        return;
      }
      if (tlv_type == 0xbb) {
        FUN_08000734();
        return;
      }
    }
    print(Invalid_TLV_command);
  }
}
return;
}
```

Nom du champ	Taille du champ
Type	1 Octet
Length	2 Octets

Value	Selon la partie Length

Type de commande	Description
1	Demande d'allocation mémoire
2	Copier la partie Value de la commande TLV dans le tampon alloué
3	Libération de l'allocation mémoire
0xcc	Faire crasher le système (disponible qu'en mode DEBUG)
0xbb	Dump des registres (disponible qu'en mode DEBUG)
0x42	Dump du flag et sortir du mode TLV
0xff	Sortir du mode TLV

5. Vulnérabilités

5.1 Heap Overflow

CWE-122: Heap-based Buffer Overflow

Un Heap overflow se produit lorsqu'un programme écrit plus de données dans un bloc de mémoire alloué dynamiquement (sur le tas) que sa taille allouée. Un débordement de tas peut entraîner la corruption des structures de données de tas adjacentes et peut entraîner l'exécution de code arbitraire au nom de l'attaquant.[2]

Dans notre cas, un dépassement de tas peut se produire, car la commande qui permet l'allocation de mémoire de taille X (commande de type 1) est complètement séparée de la commande qui permet la copie d'une valeur vers cet emplacement mémoire (commande numéro 2). Ainsi, après avoir demandé une allocation de mémoire de taille X, rien n'empêche l'utilisateur de demander de copier un contenu de taille X+ n dans cette zone mémoire, entraînant un dépassement.

```
  if (tlv_type == 1) {
    *ptr_to_malloc_length = tlv_length;
    malloc_result = malloc(tlv_length);
    *ptr_to_malloc_result = malloc_result;
    if (malloc_result != 0) {
      print([+]_Memory_allocated);
      return;
    }
    print([-]_Allocation_failed);
    return;
  }
  if (tlv_type == 2) {
    if (*ptr_to_malloc_result != 0) {
      buffer_copy(*ptr_to_malloc_result,tlv_buffer + 3,tlv_length);
      print([+]_Heap_overflow_triggered);
      return;
    }
    print([-]_No_allocated_memory);
    return;
  }
}
```

[2] https://www.packetlabs.net/posts/demystifying-overflow-attacks/

5.2 Use After Free

CWE-416: Use After Free

Cette vulnérabilité se produit lorsque du code réutilise de la mémoire après sa libération. Par la suite, la mémoire peut être à nouveau allouée et sauvegardée dans un autre pointeur, tandis que le pointeur d'origine référence un emplacement situé quelque part dans la nouvelle allocation. Toute opération utilisant le pointeur d'origine n'est plus valide, car la mémoire « appartient » au code qui opère sur le nouveau pointeur. Cette vulnérabilité peut entraîner une valeur inattendue, un crash, ou une exécution de code [3]

Dans notre cas, lorsqu'un utilisateur demande de libérer une allocation mémoire (TLV de Type 3), le pointeur *ptr_to_malloc_length n'est pas réinitialisé et continue de pointer vers la même zone mémoire, donc rien n'empêche l'utilisateur d'utiliser un TLV de Type 2 et de demander de copier du contenu vers la zone mémoire libérée.

```
if (tlv_type == 3) {
  free(*ptr_to_malloc_result);
  print([+]_Memory_freed);
  return;
}
if (tlv_type < 4) {
  if (tlv_type == 1) {
    *ptr_to_malloc_length = tlv_length;
    malloc_result = malloc(tlv_length);
    *ptr_to_malloc_result = malloc_result;
    if (malloc_result != 0) {
      print([+]_Memory_allocated);
      return;
    }
    print([-]_Allocation_failed);
    return;
  }
  if (tlv_type == 2) {
    if (*ptr_to_malloc_result != 0) {
      buffer_copy(*ptr_to_malloc_result,tlv_buffer + 3,tlv_length);
      print([+]_Heap_overflow_triggered);
      return;
    }
    print([-]_No_allocated_memory);
    return;
  }
```

[3] https://cwe.mitre.org/data/definitions/416.html

5.3 Vulnérabilité aux attaques temporelles (*Timing attack*)

CWE-208: Observable Timing Discrepancy

Cette vulnérabilité se produit lorsque deux opérations distinctes dans un code nécessitent des durées différentes pour être exécutées, d'une manière qui est observable pour un acteur et révèle des informations pertinentes pour la sécurité sur l'état du produit, par exemple si une opération particulière a réussi ou non.[4]

Dans notre cas, la fonction qui compare le mot de passe saisi par l'utilisateur avec le vrai mot de passe reviendra immédiatement à la fonction appelante après avoir rencontré un caractère invalide, et au contraire, exécutera une boucle avec un nombre potentiellement important d'itérations après chaque comparaison réussie. De cette façon, un attaquant peut essayer de saisir différentes entrées et mesurer le temps jusqu'à ce qu'il reçoive un message d'erreur pour savoir si peut-être au moins une partie du mot de passe qu'il a saisi est correct.

```
int FUN_08000830(int buffer_addr_1,int buffer_addr_2)

{
  uint current_char;
  int i;

  for (i = 0; (current_char = (uint)*(byte *)(buffer_addr_1 + i), current_char != 0 &&
              (*(byte *)(buffer_addr_2 + i) != 0)); i = i + 1) {
    if (current_char != *(byte *)(buffer_addr_2 + i)) {
      return 1;
    }
    FUN_08002d44(0x32);
  }
  i = current_char - *(byte *)(buffer_addr_2 + i);
  if (i != 0) {
    i = 1;
  }
  return i;
}
```

[4] https://cwe.mitre.org/data/definitions/208.html

5.4 Divulgation de trop d'informations en mode DEBUG

CWE-1295: Debug Messages Revealing Unnecessary Information

CWE-215: Insertion of Sensitive Information Into Debugging Code

Dans notre cas, le mode **DEBUG** accorde à l'utilisateur des autorisations de grande portée (la possibilité de visualiser le contenu des registres et même de provoquer un crash du système), son activation est trop simple (en saisissant au moins un mot de passe incorrect puis en saisissant le mot de passe **DEBUG123**). De plus, ce mode révèle complètement le vrai mot de passe à l'utilisateur.

```
void FUN_080007c4(int buffer_ptr)

{
  uint i;

  for (i = 0; i < 7; i = i + 1) {
    *(byte *)(buffer_ptr + i) = *(byte *)(DAT_08000800 + i) ^ 0x5a;
  }
  *(undefined *)(buffer_ptr + 7) = 0;
  if (*DebugModEnabled == '\x01') {
    print([DEBUG]_Decrypted_password);
    print(buffer_ptr);
    print(Entering_TLV_command_mode);
  }
  return;
}
```

6. Interaction avec le Microcontrôleur

6.1 La commande Screen

En branchant le microcontrôleur au port usb de l'ordinateur nous le retrouvons sous le nom de périphérique ttyACM0.

dmesg

```
[  140.744059] usb 2-2: new full-speed USB device number 3 using ohci-pci
[  141.051945] usb 2-2: New USB device found, idVendor=0483, idProduct=3748, bcdDevice= 1.00
[  141.051958] usb 2-2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[  141.051962] usb 2-2: Product: STM32 STLink
[  141.051965] usb 2-2: Manufacturer: STMicroelectronics
[  141.051968] usb 2-2: SerialNumber: µ[Z\x1a
[  143.988350] usb 2-3: new full-speed USB device number 4 using ohci-pci
[  144.343305] usb 2-3: New USB device found, idVendor=0483, idProduct=5740, bcdDevice= 2.00
[  144.343312] usb 2-3: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[  144.343313] usb 2-3: Product: Hack Me if you can
[  144.343314] usb 2-3: Manufacturer: GotoHack
[  144.343315] usb 2-3: SerialNumber: 304B35653035
[  144.432448] cdc_acm 2-3:1.0: ttyACM0: USB ACM device
[  144.432482] usbcore: registered new interface driver cdc_acm
[  144.432483] cdc_acm: USB Abstract Control Model driver for USB modems and ISDN adapters
[  298.659549] usb 2-3: USB disconnect, device number 4
```

Dans un premier temps nous avons interagi à l'aide de la commande screen en se connectant avec le mot de passe DEBUG123 :

sudo screen /dev/ttyACM0 115200

6.2 Script Python

Avec screen nous ne pouvions pas communiquer en TLV. Nous avons donc fait un script python. Dans un premier temps nous nous connectons automatiquement :

```
31 def main():
32         with serial.Serial(PORT, BAUD, timeout=TIMEOUT) as ser:
33                 time.sleep(5)   # Allow time for device to initialize
34
35                 print("[*] Sending password...")
36                 ser.write(b'DEBUG123\n')        # simulate typed password + newline
37
38                 time.sleep(2)
39                 response = ser.read_all()
40
41                 print("[*] Response:")
42                 print(response.decode(errors='ignore'))
43
44                 print("[*] Sending password...")
45                 ser.write(b'DEBUG123\n')        # simulate typed password + newline
46
47                 time.sleep(2)
48                 response = ser.read_all()
```

Puis faisons de multiples appels à la fonction send_tlv, qui envoi des paquets TLV et affiche les réponses :

```
 9 def send_tlv(ser, tag, value=b''):       #Manually craft and send a TLV packet
10         length = len(value)
11         packet = bytes([tag, length]) + value
12         print("value")
13         print(value)
14         ser.write(packet)
15         #print(f"⇒ Sent TLV: tag=0x{tag:02X}, len={length}, value={value}")
16
17         time.sleep(10)
18         resp = ser.read_all()
19         if resp:
20                 try:
21                         print("≤ Response:")
22                         print(resp.decode(errors='ignore'))
23                 except:
24                         print("≤ [Binary Data]")
25                         print(resp)
26         else:
27                 print("≤ No response")
```

Une fois les paquets TLV envoyés nous pouvons envoyés des strings écrites par l'utilisateur directement dans le cmd :

```python
        while True: # Now the connection stays open to send manual input
            cmd = input(">> ")
            if cmd.strip().lower() in ["exit", "quit"]:
                break
            ser.write((cmd + "\n").encode())
            time.sleep(0.1)
            resp = ser.read_all()
            print(resp.decode(errors='ignore'))
```

Exemple d'utilisation :

```
┌──(myenv)─(clem㉿kali)-[~/Documents/reverse/exam/exo2]
└─$ sudo myenv/bin/python3 script-ASM.py
[*] Sending password ...
[*] Response:

Access denied. Try again
Enter password:
[*] Sending password ...
[*] Response:

Access granted! Enter TLV commands.
Entering TLV command mode ...

value
b'0'
≤ Response:
0
value
b'10'
≤ Response:
0
value
b'01111110'
≤ Response:
0
EXIT :
value
b'11111110'
≤ Response:
0
value
b'1111111111111111'
≤ Response:
1
value
b'111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111110'
≤ Response:
10
>> exit
```

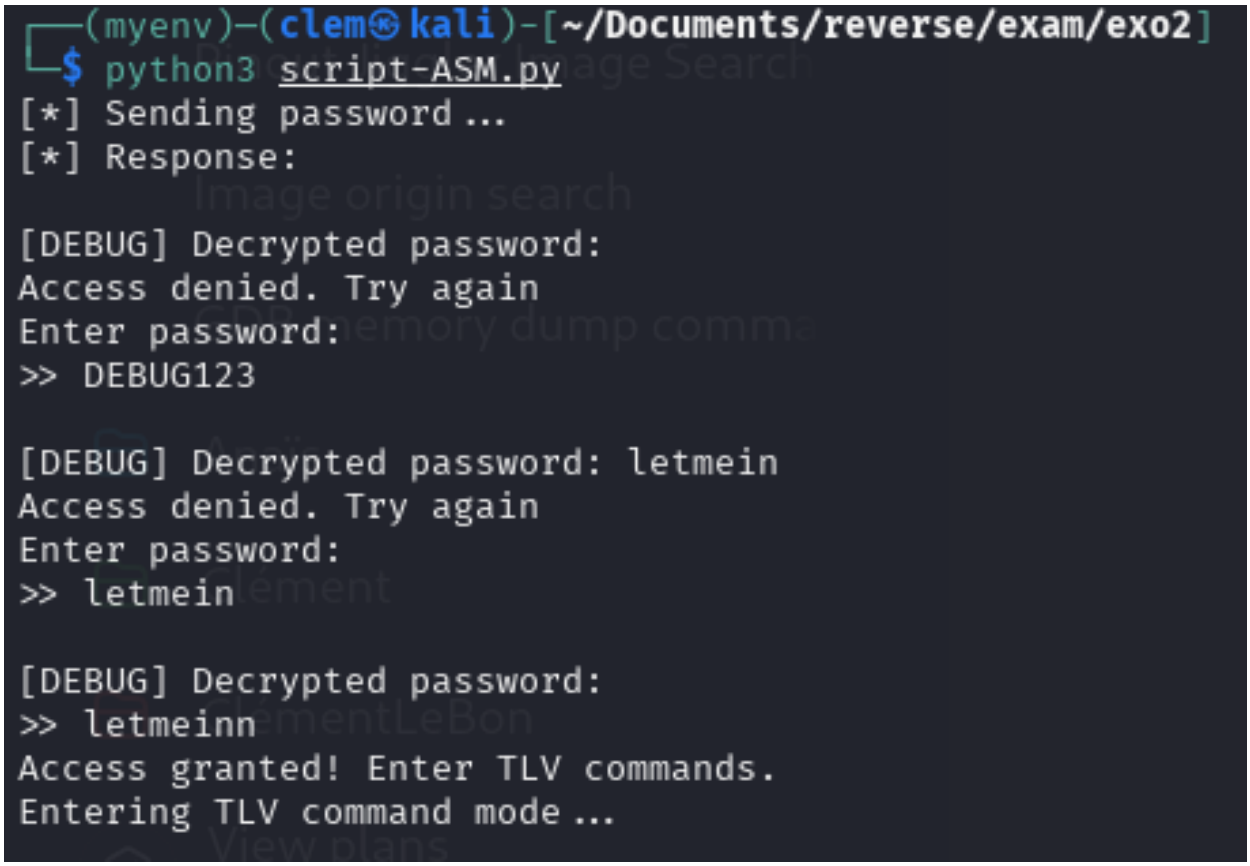
Cependant nous n'avions pas trouvé la valeur à envoyer, nous avons donc fait des tests et remarquons que les réponses retournées sont le ou les derniers caractères de la valeur.

6.3 Exploitation du mode DEBUG

Une fois que nous avons recueilli toutes les informations sur le mode DEBUG, nous pouvons accéder au mode TLV en suivant ces étapes :

- Saisir un mot de passe incorrect.
- Entrer le mot de passe DEBUG123 pour activer le mode DEBUG.
- Saisir à nouveau un mot de passe incorrect.

Le vrai mot de passe est alors affiché à l'utilisateur, car en mode DEBUG, le mot de passe est révélé lorsqu'il est chargé dans le tampon.



6.4 Flag

Flag dans le mauvais format

En exécutant plusieurs fois le script d'affilée nous avons réussi à obtenir le flag mal formé (un xor était effectué avec la valeur 0x42 au lieu de 0x5A). Nous avions eu aussi cette même string en utilisant la commande screen.

Cela nous donne : [L^cP}yhWn}j~twoe.

Ce script permet la connexion automatique (avec le mot de passe letmein) et essaye de communiquer en TLV en utilisant le type 0x42 afin d'envoyer des commandes.

```
┌──(myenv)─(clem㉿kali)-[~/Documents/reverse/exam/exo2]
└─$ sudo myenv/bin/python3 script-ASM.py
[*] Sending password...
[*] Response:

[DEBUG] TLV Received - Type: 0x42, Length: 12289
[L^cP}yhWn}j~twoe
Enter password:
[*] Sending password...
[*] Response:

Access granted! Enter TLV commands.
Entering TLV command mode...

value
b'0'
⩽ Response:
0
value
b'10'
⩽ Response:
0
```

Flag dans le bon format

L'unique moment où nous avons utilisé le fichier ELF fourni a été pour trouver le flag. En allant dans la fonction DecryptFlag nous remarquons que la valeur dans la ram à l'adresse 0x20000000 et la valeur 0x42 sont utilisées pour faire un xor et que le résultat est le flag déchiffré.

```
 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) ^ 0x42;
11   }
12   output[0x11] = '\0';
13   return;
14 }
```

En allant à l'adresse 0x20000000, grâce au dump de la ram effectué plus tôt nous trouvons cette string.

```
20000000 19 0e 1c        uint8_t[...  19h,0Eh,1Ch,"!",12h,"?;*",15h,",?(<65-'"   ; main.c:61
         21 12 3f
         3b 2a 15 ...
```

Cependant en effectuant un xor en utilisant 0x42 avec chaque caractère cela nous donne :

[L^cP}yhWn}j~twoe

Nous avons donc décidé de brute force la valeur avec laquelle il faut exécuter le xor :

```python
tab = [
    0x19, 0x0E, 0x1C, ord('!'), 0x12, ord('?'), ord(';'), ord('*'),
    0x15, ord(','), ord('?'), ord('('), ord('<'), ord('6'), ord('5'),
    ord('-'), ord("'")
]

for key in range(1, 256):
    try:
        decoded = ''.join(chr(b ^ key) for b in tab)
        if '{' in decoded and '}' in decoded:
            print(f"[KEY 0x{key:02X}] {decoded}")
    except:
        continue
```

Cela nous donne :

```
┌──(myenv)─(clem㉿kali)-[~/Documents/reverse/exam/exo2]
└─$ python3 decrypt.py
[KEY 0x51] H_MpCnj{D}nymgd|v
[KEY 0x57] NYKvEhl}B{hkabzp
[KEY 0x5A] CTF{HeapOverflow}
[KEY 0x5C] ER@}NcgvIpct`jiq{
```

Nous remarquons une string intéressante : CTF{HeapOverflow}

C'est bien le flag !

6.5 Exploitation d'une vulnérabilité

Nous avons tenté d'exploiter la vulnérabilité de heap overflow à l'aide du script python. En reprenant les fonctions vues ci-dessus, nous nous sommes connectés puis nous avons envoyé un premier paquet TLV afin d'allouer une taille de buffer de 32 bits. Ensuite, nous avons envoyé un deuxième paquet afin de remplir le buffer avec un payload de 128 bits, dépassant ainsi la taille du buffer. Cependant nous n'avons pas eu le temps de tester ce code.

```
47          # STEP 1: Allocate a small buffer (Type 1, Length 32)
48          print("[*] Allocating 32-byte heap buffer...")
49          send_tlv(ser, 0x01, b'\x00' * 32)
50          
51
52          # STEP 2: Trigger overflow with 128-byte payload (Type 2, Length 128)
53          print("[*] Sending overflow payload (128 bytes)...")
54          overflow_payload = b'A' * 128
55          send_tlv(ser, 0x02, overflow_payload)
```

Bibliographie

GLiNet :

- https://gzhls.at/blob/ldb/3/5/7/2/574814232500cbfeedc718795d96c2783748.pdf
- https://images-na.ssl-images-amazon.com/images/I/B1oCL2RA8YS.pdf
- https://www.gl-inet.com/products/gl-mt300n-v2/

Datasheet MT7628**:**

- https://www.alldatasheet.com/datasheet-pdf/view/1131988/ETC2/MT7628.html?ref=boschko.ca

Datasheet Winbond :

- https://www.lcsc.com/datasheet/lcsc_datasheet_2412251211_Winbond-Elec-W25Q128JVSIQ_C97521.pdf

Micron D9RZH :

- https://www.worldwayelec.com/pro/micron/d9rzh/3866633
- https://www.micron.com/products/memory/dram-components/ddr2-sdram/part-catalog

Programming manual PM0214 :

- https://www.st.com/resource/en/programming_manual/pm0214-stm32-cortexm4-mcus-and-mpus-programming-manual-stmicroelectronics.pdf

Datasheet STM32F40XXX :

- https://www.st.com/resource/en/datasheet/dm00037051.pdf

STM32F405.svd :

- https://github.com/modm-io/cmsis-svd-stm32/blob/main/stm32f4/STM32F405.svd

Annexe

Code 1

Version 1 du code pour exploiter le heap overflow, avec des sleep :

```
import serial
import time

PORT = '/dev/ttyACM0'
BAUD = 115200
TIMEOUT = 1

def send_tlv(ser, tag, value=b''):
    length = len(value)
    length_bytes = length.to_bytes(2, byteorder='little')
    packet = bytes([tag]) + length_bytes + value
    print(f"=> Sent TLV: tag=0x{tag:02X}, len={length}")
    ser.write(packet)

    time.sleep(1)
    resp = ser.read_all()
    if resp:
```

```
        try:
            print("<= Response:")
            print(resp.decode(errors='ignore'))
        except:
            print("<= [Binary Data]")
            print(resp)
    else:
        print("<= No response")

def main():
    with serial.Serial(PORT, BAUD, timeout=TIMEOUT) as ser:
        # Let the device initialize
        time.sleep(3)

        # Authenticate with password
        print("[*] Sending password...")
        ser.write(b'letmein\n')
        time.sleep(2)
        response = ser.read_all()
        print("[*] Response:")
        print(response.decode(errors='ignore'))

        # Send password twice
        print("[*] Sending password...")
```

```
    ser.write(b'letmein\n')
    time.sleep(2)
    response = ser.read_all()
    print("[*] Response:")
    print(response.decode(errors='ignore'))

    # STEP 1: Allocate a small buffer (Type 1, Length 32)
    print("[*] Allocating 32-byte heap buffer...")
    send_tlv(ser, 0x01, b'\x00' * 32)


    # STEP 2: Trigger overflow with 128-byte payload (Type 2, Length 128)
    print("[*] Sending overflow payload (128 bytes)...")
    overflow_payload = b'A' * 128
    send_tlv(ser, 0x02, overflow_payload)


if __name__ == '__main__':
  main()
```

Code 2

Version 2 : nous avons modifié le code afin de ne plus utiliser sleep mais des expect. Voici le code final pour exploiter le heap overflow :

```
import pexpect
from pexpect_serial import SerialSpawn

PORT = '/dev/ttyACM0'
BAUD = 115200

def send_tlv(ser, tag, value=b'', expect=None):
    length = len(value)
    length_bytes = length.to_bytes(2, byteorder='little')
    packet = bytes([tag]) + length_bytes + value
    ser.write(packet)
    print(f"=> Sent TLV: tag=0x{tag:02X}, len={len(value)}")

    if expect:
        try:
            ser.expect(expect, timeout=3)
            print("<= Expected response:")
            print(ser.before.decode(errors='ignore') + ser.after.decode(errors='ignore'))
        except pexpect.TIMEOUT:
            print(f"<= Timeout waiting for: {expect}")
```

```
        except Exception as e:
            print("<= Error:", e)
    else:
        try:
            response = ser.read_nonblocking(size=1024, timeout=1)
            print(response.decode(errors='ignore'))
        except:
            print("[No immediate response]")


def login(ser):
    for attempt in [1, 2]:
        print(f"[*] Login attempt {attempt} with 'letmein'")
        ser.expect("Enter password:", timeout=3)
        ser.sendline("letmein")
        ser.expect("Access granted!", timeout=3)
        print("[+] Authenticated.")


def main():
    ser = SerialSpawn(PORT, baudrate=BAUD, timeout=1)

    # Perform login twice
    login(ser)

    # Step 1: Allocate 32-byte buffer
```

```
    print("[*] Allocating heap...")
    send_tlv(ser, 0x01, b'\x00' * 32, expect="Memory allocated")

    # Step 2: Overflow with 128 bytes
    print("[*] Sending overflow payload...")
    overflow_payload = b'A' * 128
    send_tlv(ser, 0x02, overflow_payload, expect="Heap overflow triggered")

if __name__ == '__main__':
    main()
```