

Challenge SSTIC 2014

Résolution

Remerciements :-)

*Je tenais d'abord à remercier les concepteurs de ce challenge, hautement addictif et éducatif.
Mais aussi et surtout ma femme et mes enfants qui n'auront pas souvent pu profiter de ma
présence dans les moments de concentration intense dans un emploi du temps bien chargé !*

Chapitre 1 - Analyse de la trace USB

Le fichier fourni est au format XZ, format de compression ouvert.

Il s'agit, une fois décompressé, d'un fichier texte résultant du dump d'une trace USB envoyée par mail.

Cette trace, au format texte, ne peut être importée en l'état dans un outil (tel que Wireshark par exemple). Mais nous pouvons réaliser quelques scripts pour automatiser son analyse.

Nous pouvons reconnaître un périphérique USB communiquant sur plusieurs canaux, et notamment, sur le canal '008', une communication ADB Android.

Nous pouvons donc mettre en place un script affichant ces échanges¹.

Après avoir vérifié l'utilisateur, le système et le contenu des répertoires '/sdcard', '/sdcard/Documents' et '/data/local/tmp', nous observons l'upload du fichier :
'/data/local/tmp/badbios.bin'.

Le nom du fichier est peut-être une référence à un hypothétique malware nommé BadBIOS, connu pour se propager et communiquer avec ses homologues par les hauts-parleurs et micro des PC alentours en ultrason.

Le protocole ADB, après avoir donné le nom du fichier et ses permissions en décimal, fournit le contenu de celui-ci par bloc d'au plus 64 Ko, précédé du header "DATA" et de la taille du bloc sur 4 octets. Nous avons donc 2 blocs, le premier de 64 Ko et le second de 12 464 octets, pour obtenir au final un fichier de 78 000 octets.

¹ Annexe A : script dump USB canal 008

Chapitre 2 - Analyse du fichier badbios.bin

Nous pouvons constater qu'il pourrait s'agir d'un fichier exécutable ARM 64 bits

```
$ file badbios.bin
badbios.bin: ELF 64-bit LSB executable, version 1 (SYSV), statically linked, stripped

$ objdump -h badbios.bin

badbios.bin:  file format elf64-littleaarch64
```

```
Sections:
Idx Name          Size      VMA           LMA           File off  Algn
  0 .text          0000048c  000000000001010c  000000000001010c  0000010c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata        00000040  0000000000010598  0000000000010598  00000598  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data          00011f50  0000000000021000  0000000000021000  00001000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
```

On pourra aussi demander à objdump de réaliser le désassemblage de la section text.

Vu le faible nombre d'instructions employées, et afin de bien comprendre le fonctionnement du code exécuté, il est possible de s'orienter sur un simulateur ARM réalisé manuellement². Celui-ci va nous montrer que le code va réaliser 2 mmap à des adresses fixes, les remplir chacun en utilisant le contenu du segment data et positionner les droits pour obtenir un nouveau segment de code et un de données.

Suite à la création de ces segments, plus gros que le segment de données originel, nous avons un jump dans le nouveau segment de code. A titre de comparaison, si nous compressons les segments obtenus, nous obtenons à peu près la taille du premier segment de données. Nous pouvons donc supposer qu'il s'agit d'une étape de décompression, et que le code réel se trouve dans les nouveaux segments.

Nous ne chercherons pas ici à comprendre le processus de décompression, mais nous allons dumper dans notre simulateur les segments obtenus et les analyser à leur tour.

² Annexe B : Simulateur ARM 64

Chapitre 3 - badbios.bin, sections décompressées

Cette fois-ci, nous pourrions reprendre le simulateur et sauvegarder l'usage, à chaque instruction, des échanges de données. Cependant l'exécution montre que le fil d'exécution est bien trop important et génère des dizaines de Go de logs, difficilement analysables. L'approche va donc plutôt reprendre une analyse statique plus standard du code exécuté.

En mettant en place un script fournissant le code associé à une adresse de fonction³, qui va ainsi ressortir l'ensemble du code (y compris suite à un jump conditionnel) jusqu'à un ret - et optionnellement le désobfusquer s'il y a lieu -, nous pouvons décompiler les fonctions principales. Afin de réaliser le désassemblage, nous utiliserons cette fois "capstone".

Le code principal et d'initialisation est assez court (ici en pseudo-code manuel) :

```
fct_400514:
    exit(fct_4000b0())

fct_4000b0:
    u64 var
    if (fct_402960(mmap50, 16, &var) < 0)
        return -1
    else
        return fct_402914(var, 16)

fct_402960(x0, x1, x2) {
    if ((r1 = mmap(0, 4096, 3, 0x22, 0, 0)) == -1) : return
    if ((r1[0x50] = mmap(0, x1 + 4095 & 0xff..ff000, 3, 0x22, 0, 0)) == -1) : return
    for (i = 0; i < 0x20; i++)
        r1[0x68 + i*24] &= 0xffffffffe
        r1[0x58 + i*24] = 0
        r1[0x60 + i*24] = 0
    if ((r3 = mmap(0, 4096, 3, 0x22, 0, 0))) == -1)
        r1[0x358] = 0
    else
        if ((r1[0x358] = r3) != 0)
            r4 = fct_401a08(r1)
            fct_400404()
            fct_400408(r1 + 0x10, 0x510000)
            fct_40047c(r1 + 0x10, 0x510000 + 0x20)
            [r1 + 4] = 0
            ul([r1]) &= 0xffffffffe
            [r1 + 8] = 0
            if (x1 != 0)
                for(i = 0; i < x1; i++): x0[i] = [r1[0x50]][i]
            [x2] = r1
        return 0
    return -1
}
```

La fonction initialise 32 blocs de données et quelques structures fraîchement allouées.

³ Annexe C : Extraction de fonction

En suivant les fonctions suivantes, on tombe alors sur les fct_400404 et fct_400498, dont la complexité vient essentiellement de la quantité de calculs qui y sont réalisés.

Un bloc de 64 octets y est effectivement traité, copié dans la pile puis manipulé par 16 registres (de 32 bits). Ces opérations effectuent en série, tour après tour, une addition, un xor, une rotation à droite de (resp.) 16, 20, 24 et 25 bits⁴, etc.

Ces opérations, combinées à une chaîne de caractères utilisée dans une des fonctions d'initialisation (fct_400408) : "expand 16-byte k", nous permettent de supposer l'utilisation de l'algorithme de cryptage "chacha", variante du "salsa20".

En reprenant son implémentation classique, nous pouvons alors mapper certaines fonctions :

```
// ECRYPT_init
fct_400404() {
}

// ECRYPT_keysetup(ECRYPT_ctx *x, const u8 *k, u32 kbits, u32 ivbits)
fct_400408(x0, x1) {
    [x0 + 0x10] = [x1]
    [x0 + 0x14] = [x1 + 4]
    [x0 + 0x18] = [x1 + 8]
    [x0 + 0x1c] = [x1 + 0xc]
    [x0 + 0x20] = [x1]
    [x0 + 0x24] = [x1 + 4]
    [x0 + 0x28] = [x1 + 8]
    [x0 + 0x2c] = [x1 + 0xc]
    [x0] = 0x61707865 // "expa"
    [x0 + 4] = 0x3120646e // "nd 1"
    [x0 + 8] = 0x79622d36 // "6-by"
    [x0 + 0xc] = 0x6b206574 // "te k"
    return x0
}

// ECRYPT_ivsetup(ECRYPT_ctx *x, const u8 *iv)
fct_40047c(x0, x1) {
    [x0 + 0x30] = 0
    [x0 + 0x34] = 0
    [x0 + 0x38] = [x1]
    [x0 + 0x3c] = [x1 + 4]
}

// ECRYPT_encrypt_bytes(ECRYPT_ctx *x, const u8 *m, u8 *c, u32 bytes)
fct_400498(x0, x1, x2, x3)
fct_4004a4(x0, x1, x2, x3)
```

⁴ L'algorithme "chacha" se base en réalité sur une rotation à gauche de (resp.) 16, 12, 8 et 7 bits, mais l'assembleur ARM ayant implémenté une rotation à droite, ces valeurs sont équivalentes sur 32 bits.

Maintenant que nous avons ces fonctions, nous pouvons voir comment elles sont employées. Notamment la fonction `fct_4020c4`, prenant un offset en paramètre (en plus de la structure de contexte du chiffage), se charge de décrypter le bloc de 64 octets correspondant (qui la contient) et renvoie l'adresse du bloc clair une fois terminé.

Il s'agit d'une des fonctions clefs, sur laquelle se base un certain nombre d'autres fonctions manipulant la mémoire. Cette fonction se base sur des blocs mémoires permettant de ne pas figer une adresse physique pour un offset spécifique (et ainsi rendre plus difficile une analyse dynamique).

On pourra notamment remarquer :

`fct_4022ac(ctx, off, buf, len)`

Décrypte 'len' octets à partir de l'offset 'off', en déplaçant le résultat dans 'buf'

`fct_402364(ctx, off, buf, len)`

Encrypte 'len' octets à l'offset 'off', depuis les valeurs situées dans 'buf'

`fct_4026cc(ctx)`

Décrypte la valeur 32 bits située à l'offset 60 (0x3c)

`fct_4025f4(ctx, n)`

Décrypte la valeur 32 bits située à l'offset $(n - 1) * 4$

`fct_402660(ctx, n, val)`

Encrypte 'val' (32 bits) à l'offset $(n - 1) * 4$

Par ailleurs, si on modifie - *volontairement* - le comportement du simulateur ARM pour y introduire - *volontairement* - quelques bugs, le programme nous renvoie des messages du type "Invalid instruction". Dans l'exécutable, on trouve aussi la chaîne "Bad instruction pointer". Ce type de message pourrait nous laisser entendre que le programme exécute du pseudo-code par l'intermédiaire d'une VM.

Si c'est bien le cas, nous devrions trouver une boucle principale, basée sur une lecture de la position en cours (généralement appelé registre PC), lecture du pseudo-code correspondant, déplacement du PC, d'un tableau de fonctions (ou équivalent) et appel du code associé.

Or dans la fonction `fct_402754`, nous pouvons justement constater une boucle avec

- Appel de `fct_4026cc` - qui retourne la valeur à l'offset `0x3c` - (résultat en `x20`)

```
mov x0, x19
bl #0x4026cc
mov x20, x0
```

- Lecture d'un seul octet à l'adresse `x20` par `fct_4022ac`

```
mov x1, x0
add x2, x29, #0x5c ; buffer dans la pile
mov x0, x19
movz x3, #1
bl #0x4022ac
```

- Vérification que la valeur de cet octet est bien comprise entre `0x00` et `0x1e`

```
ldrb w3, [x29,#0x5c]
cmp w3, #0x1f
b.hi #0x402888 ; Instruction invalide
```

- S'il est inférieur à 8, relecture de 4 - sinon de 2 - octets à la même adresse `x20`

```
movz x25, #2
movz x24, #4

cmp w3, #8
csel x22, x25, x24, hi
mov x3, x22
bl #0x4022ac
```

- Utilisation d'un tableau de pointeur pour appeler un code selon ce premier octet

```
ldrb w0, [x29,#0x5c]
add x0, x0, #0x6c
ldr x2, [x19,x0,ls1 #3]
mov x0, x19
blr x2
```

Nous en déduisons donc que le PC est situé à l'offset `60` (`0x3c`), et que chaque accès en lecture comme en écriture dans la VM fait l'objet d'un cryptage pour laisser une empreinte lisible minimale pendant son exécution.

Nous pouvons dumper le tableau de pointeur quand nous atteignons cette fonction.

Aussi, les données étant correctement chiffrées, avec un IV incrémenté tous les 64 octets, nous nous en servons, avec le contexte de chiffrement désormais connu, pour obtenir l'ensemble de ces données en clair.

Chapitre 4 - Etude de la VM

Instructions sur 4 octets		Instructions sur 2 octets	
00:	0x00400d9c	09:	0x00400d58
01:	0x00400dac	0a:	0x00400c90
02:	0x00401580	0b:	0x00400c20
03:	0x00401634	0c:	0x00400bd0
04:	0x004016e4	0d:	0x00400b78
05:	0x00401030	0e:	0x00400b04
06:	0x004010ec	0f:	0x00400a8c
07:	0x004011b4	10:	0x00400a08
08:	0x00401794	11:	0x00400978
		12:	0x00400918
		13:	0x004008c4
		14:	0x00400864
		15:	0x004007ec
		16:	0x00400d24
		17:	0x00400ce0
		18:	0x00401970
		19:	0x004018d0
		1a:	0x0040187c
		1b:	0x004005f4
		1c:	0x004005fc
		1d:	0x00401490
		1e:	0x0040077c

Tableau de fonctions pour les opcodes de la VM

Début de la mémoire de la VM :

```

0x00000000: 00000000 00000000 00000000 00000000 .....
0x00000010: 00000000 00000000 00000000 00000000 .....
0x00000020: 00000000 00000000 00000000 00000000 .....
0x00000030: 00000000 00200000 00000000 40000000 .....@...
0x00000040: 00010000 01210000 00020000 01120000 .....!.....
0x00000050: 00030000 01E33200 00040000 01440200 .....2.....D..
0x00000060: 1D000001 00000111 00000A22 00030000 .....".
0x00000070: 01C33F00 00040000 01040100 1D000205 ..?.
0x00000080: 00000003 00000103 01001335 086AB402 .....5.j..

```

On y voit notamment que le PC est au départ à la valeur de 0x00000040, situé en suivant. La première pseudo-instruction est donc "00 01 00 00", puis "01 21 00 00", etc.

On notera par la suite quelques chaînes de caractères telles que

```

:: Please enter the decryption key:
:: Trying to decrypt payload...
  Wrong key format.
  Invalid padding.
  Cannot open file payload.bin.
:: Decrypted payload written to payload.bin.
payload.bin

```

Par ailleurs une grande partie de la pseudo-mémoire est remplie de '00', mais on remarquera une zone entre 0x00008000 et 0x0000a0000 remplie de valeurs aléatoires (ou cryptées).

A chaque appel de fonction pour un pseudo-code, ce dernier est passé en entier (2 ou 4 octets) en paramètre à la fonction (avec le contexte de chiffrement). Nous allons donc maintenant pouvoir analyser le code des pseudo-instructions utilisées.

VM : 0x00

	v	r	Opcode
4	16	4	8

Le pseudo-code est découpé en 3 parties, l'opcode lui-même (8 bits) désormais inutilisé, un index (4 bits) et une valeur (16 bits). Cette valeur est décalée de 16 bits sur la gauche.

```
ubfx x2, x1, #0xc, #0x10
lsl w2, w2, #0x10

ubfx x1, x1, #8, #4
```

La valeur obtenue est alors placée à l'offset (index - 1) * 4

```
sub w1, w1, #1
sbfiz x1, x1, #2, #0x20
add x2, x29, #0x20
movz x3, #4
str w4, [x29, #0x20]
bl #0x402364
```

Ce code est aussi accompagné de vérifications, parfois génériques voire inutiles (comme la vérification que l'index est inférieur à 16, bien qu'il soit obtenu à partir de 4 bits).

Dans le contexte d'une VM, on peut se représenter les 15 premiers mots de 32 bits de la mémoire comme les registres de la VM, plus le PC, et cet opcode se charge donc de prendre une valeur immédiate et de remplir les 16 bits de poids forts d'un registre avec celle-ci. Nous l'appellerons donc "**mov.h**" (et les registres seront r1 à r15).

VM : 0x01

Ce pseudo-code est très proche du pseudo-code 0x00. Cependant, la valeur immédiate n'est pas décalée de 16 bits, mais est combinée par un OR avec la valeur en cours du registre. Elle permet donc de compléter l'opcode précédent afin de charger une valeur immédiate de 32 bits dans un registre de la VM en 2 pseudo-code successifs.

Nous appellerons donc celui-ci "**mov.l**" et nous simplifierons un couple mov.h/mov.l par "**mov**".

VM : 0x02, 0x03, 0x04

Ces 3 pseudo-codes fonctionnent sur le même modèle.

v	r2	r1	Opcode
16	4	4	8

Dans les trois cas, le registre r2 est lu, et son contenu additionné avec v pour former une adresse. La valeur en mémoire à cette adresse est alors lue et son contenu est mis dans r1.

La différence entre ces 3 pseudo-code se situe dans la taille de la mémoire lue :

0x02	32 bits	ld.32
0x03	16 bits	ld.16
0x04	8 bits	ld.8

VM : 0x05, 0x06, 0x07

De nouveau, ces 3 pseudo-code sont très similaires

v	r2	r1	Opcode
16	4	4	8

Cette fois-ci, il s'agit d'une opération d'écriture du registre r1 à l'adresse indiquée.

0x02	32 bits	st.32
0x03	16 bits	st.16
0x04	8 bits	st.8

VM : 0x08

offset	cond	r	C	Opcode
16	3	4	1	8

Cet opcode va permettre de réaliser l'ensemble des branchements, jump et call, y compris conditionnels. Le bit "C" va permettre de conserver le PC de l'opcode suivant dans le registre r15. Cela permettra un retour à l'appelant, on parlera donc de "call", et sinon de "b".

Les 3 bits de "cond" vont définir une condition à appliquer à r pour que le saut s'applique.

000	vrai	b	call
001	faux		
010	r == 0	b.z	
011	r != 0	b.nz	
100	r < 0	b.lt	
101	r > 0	b.gt	
110	r <= 0	b.le	
111	r >= 0	b.ge	

VM : 0x09

Il s'agit du premier opcode sur 16 bits et non plus 32.

r2	r1	Opcode
4	4	8

L'opération effectuée étant simplement $r1 = \text{not}(r2)$. Il s'agit d'une inversion binaire.

VM : 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x12, 0x13, 0x14, 0x15

r2	r1	Opcode
4	4	8

Tous ces opcodes appliquent la même construction : $r1 = r1 \text{ (op) } r2$, ces opérations étant :

0x0a	xor	0x12	+
0x0b	or	0x13	-
0x0c	and	0x14	*
0x0d	shl	0x15	/
0x0e	shr		
0x0f	ror		

VM : 0x16, 0x17

	r	Opcode
4	4	8

Pour ces opcodes, les opérations sont simplement une incrémentation pour 0x16 et une décrémentation pour 0x17.

VM : 0x1c

	Opcode
8	8

Ici, il s'agit tout simplement de provoquer l'arrêt de la VM.

VM : 0x1d

	Opcode
8	8

Ce pseudo-code permet d'effectuer des appels systèmes, selon la valeur des registres.

Selon la valeur de r1, les appels seront

0x00 : la VM effectue un "openat" sur le fichier dont le nom est pointé (dans la mémoire de la VM) par le registre r2, les flags en r3 et les droits d'accès étant indiqués par r4

r1 = openat(AT_FDCWD, r2, r3, r4)

0x01 : la VM effectue cette fois un r1 = read(r2, r3, r4)

0x02 : l'appel sera maintenant r1 = write(r2, r3, r4)

0x03 : Enfin celui-ci permet de réaliser close(r2)

VM : 0x1e

r2	r1	Opcode
4	4	8

Nous avons droit ici à une pseudo-instruction permettant le calcul de parité binaire de r2, le résultat (0 ou 1 selon que le nombre de bit à 1 dans r2 soit pair) étant positionné dans r1.

Cette analyse des pseudo-instructions permet de réaliser un pseudo-désassembleur⁵.

⁵ Annexe D : Désassembleur VM

Chapitre 5 - Code exécuté par la VM

Maintenant que nous avons vu l'ensemble du jeu d'instructions et récupéré le listing, nous pouvons analyser le code exécuté par la VM.

```
0040: mov r1, #0x00000002
0048: mov r2, #0x00000001
0050: mov r3, #0x0000032e
0058: mov r4, #0x00000024
0060: syscall                ; Affiche ":\: Please enter the decryption key:" sur
stdout
0062: mov r1, #0x00000001
006a: xor r2, r2
006c: mov r3, #0x000003fc
0074: mov r4, #0x00000010
007c: syscall                ; Lecture en 0x03fc depuis stdin
007e: mov r5, r1
0082: mov r3, #0x00000010
008a: sub r5, r3
008c: b.nz r5, 02b4          ; Jump si retour du read différent de 16
```

Si la longueur de la chaîne lue est différente de 16 octets :

```
02b4: mov r1, #0x00000002
02bc: mov r2, #0x00000002
02c4: mov r3, #0x00000374
02cc: mov r4, #0x00000015
02d4: syscall                ; Affiche " Wrong key format."
02d6: b 02b2

02b2: halt
```

Sinon nous pouvons continuer

```
0090: mov r15, #0x00000010
0098: mov r14, #0x000003fc ; Adresse du buffer d'entrée (read)
00a0: mov r13, #0x00000326
00a8: dec r13
00aa: mov r2, #0x00000030 ; '0'
00b2: mov r3, #0x00000039 ; '9'
00ba: mov r4, #0x00000041 ; 'A'
00c2: mov r5, #0x00000046 ; 'F'
00ca: ld.8 r12, r14
00ce: mov r1, r12
00d2: sub r1, r2
00d4: b.lt r1, 02b4          ; c < '0' => Erreur
00d8: mov r1, r12
00dc: sub r1, r3
00de: b.le r1, 0106         ; '0' <= c <= '9' => ok
00e2: mov r1, r12
00e6: sub r1, r4
00e8: b.lt r1, 02b4          ; c < 'A' => Erreur
00ec: mov r1, r12
00f0: sub r1, r5
00f2: b.gt r1, 02b4          ; c > 'F' => Erreur
00f6: sub r12, r4
00f8: mov r1, #0x0000000a
```

```

0100: add r12, r1
0102: b 0108
0106: sub r12, r2          ; r12 = c converti de l'hexa (ascii)
0108: mov r7, #0x00000010
0110: sub r7, r15
0112: mov r1, #0x00000001
011a: and r1, r7
011c: b.nz r1, 012c        ; quartet de bits forts => décalage
0120: mov r7, #0x00000004
0128: shl r12, r7
012a: inc r13
012c: ld.8 r1, r13
0130: or r1, r12
0132: st.8 r13, r1
0136: inc r14
0138: dec r15
013a: b.nz r15, 00ca     ; quartet suivant

```

Suite à ces instructions, nous avons à l'adresse 0x0326 une conversion de l'hexadécimal de la chaîne entrée fournie (16 caractères => 8 octets).

Nous savons donc maintenant que le programme attend 8 octets au format hexadécimal.

```

013e: mov r1, #0x00000002
0146: mov r2, #0x00000001
014e: mov r3, #0x00000354
0156: mov r4, #0x00000020
015e: syscall              ; Affiche ":: Trying to decrypt payload..."

0160: mov r1, #0x00000326
0168: ld.32 r10, r1
016c: ld.32 r11, 0x0004 + r1 ; Copie les 8 octets en r10 (poids fort) et r11 (poids
faible)

0170: xor r1, r1
0172: mov r2, #0x00008000   ; Adresse du bloc de données cryptées
017a: mov r3, #0x00000008   ; Boucle interne sur 8 bits
0182: xor r4, r4
0184: mov r12, #0xb0000000 ; Masque poids fort
018c: mov r13, #0x00000001 ; Masque poids faible

```

Nous commençons ensuite une boucle

```

0194: mov r8, r10
0198: mov r9, r11
019c: and r8, r12          ; Applique le masque sur la partie haute
019e: and r9, r13          ; Puis celui sur la partie basse
01a0: xor r8, r9
01a2: parity r9, r8        ; Calcule la parité sur le résultat

01a4: mov r8, #0x00000001
01ac: mov r7, #0x0000001f
01b4: mov r6, r10
01b8: and r6, r8
01ba: shl r6, r7
01bc: shr r11, r8
01be: or r11, r6

```

```

01c0: shr r10, r8          ; Décale r10/r11 de 1 bit vers la droite

01c2: shl r9, r7
01c4: or r10, r9          ; Le bit de poids fort de r10/r11 est le résultat de la
parité

01c6: dec r3
01c8: mov r7, r11
01cc: and r7, r8
01ce: shl r7, r3
01d0: or r4, r7          ; Le nouveau bit de poids faible est ajouté dans un masque
01d2: b.nz r3, 01f6      ; On continue tant que 8 bits n'ont pas été ainsi extraits

01d6: mov r7, #0x00008000
01de: add r7, r1
01e0: ld.8 r8, r7
01e4: xor r8, r4          ; Le masque généré est XOR'é avec le buffer en 0x8000, octet
01e6: st.8 r7, r8        ; par octet, et le résultat remplace le buffer d'origine.
01ea: mov r3, #0x00000008
01f2: inc r1
01f4: xor r4, r4

01f6: mov r8, #0x00002000
01fe: sub r8, r1
0200: b.gt 0194          ; On continue ainsi sur 8192 octets

```

Nous avons donc une boucle de décryptage, et cet algorithme, une fois la clef récupérée nous permettra de déchiffrer le bloc à l'offset 0x8000.

```

0204: mov r13, #0x00008000
020c: mov r12, #0x00002000
0214: mov r11, #0x00000080
021c: xor r10, r10
021e: mov r9, #0x00000008

0226: inc r10
0228: dec r12
022a: b.le r12, 02da
022e: mov r10, r13
0232: add r10, r12
0234: ld.8 r1, r10        ; Nous lisons le résultat en partant de la fin du buffer et
0238: b.z r1, 0226        ; continuons tant que nous avons des '\0x00' (r10 = nombre de
0)

023c: sub r1, r11          ; Le dernier octet non nul doit valoir 0x80
023e: b.nz r1, 02da      ; Sinon => erreur
0242: sub r10, r9         ; Le nombre d'octets nuls doit être supérieur à 8
0244: b.le r10, 02da     ; Sinon => erreur

```

Ces informations sont primordiales. Nous savons maintenant qu'au moins les 8 derniers octets du résultat doivent être nuls. Cela devrait être suffisant pour nous permettre de calculer la clef de décryptage.

```

0248: and r1, r0
024a: mov r2, #0x000003f0
0252: mov r3, #0x00000241
025a: mov r4, #0x000001b6
0262: syscall ; Ouverture du fichier "payload.bin" en écriture
0264: b.lt r1, 02b2 ; Halt si erreur

0268: mov r2, r1
026c: mov r1, #0x00000002
0274: mov r3, #0x00008000
027c: mov r4, r12
0280: syscall ; Ecriture du buffer décrypté jusqu'à l'octet non nul
non compris

0282: mov r1, #0x00000003
028a: syscall ; Fermeture du fichier

028c: mov r1, #0x00000002
0294: mov r2, #0x00000001
029c: mov r3, #0x000003c2
02a4: mov r4, #0x0000002d
02ac: syscall ; Affiche ":: Decrypted payload written to
payload.bin."
02ae: b 02b2

02b2: halt

```

Ensuite un cas d'erreur avec son message associé.. mais non utilisé. Il aurait fallu pour cela que l'instruction en 0264 soit `b.lt r1, 0300`.

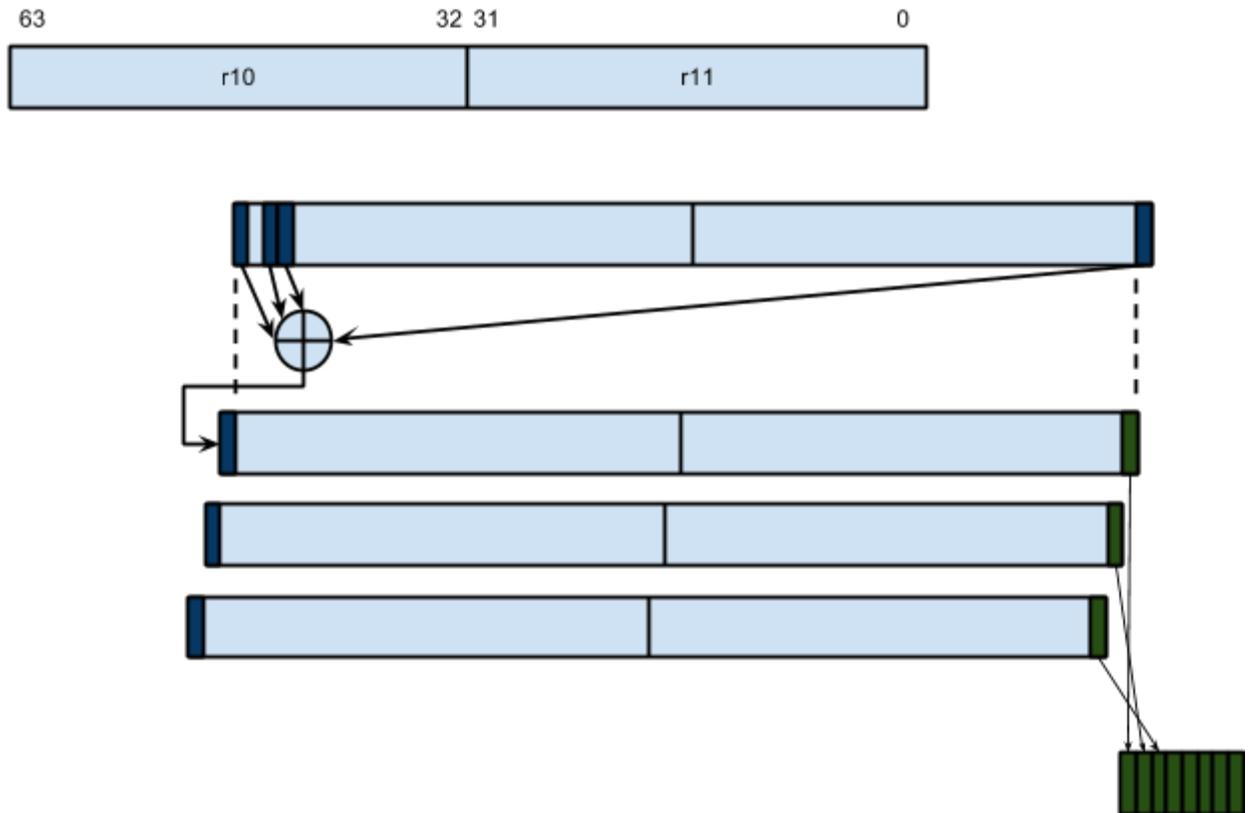
```

0300: mov r1, #0x00000002
0308: mov r2, #0x00000002
0310: mov r3, #0x000003a0
0318: mov r4, #0x00000021
0320: syscall ; Affiche " Cannot open file payload.bin."
0322: b 02b2 ; Quitte

```

Chapitre 6 - Récupération de la clef

L'algorithme de décryptage employé dans la VM est le suivant :



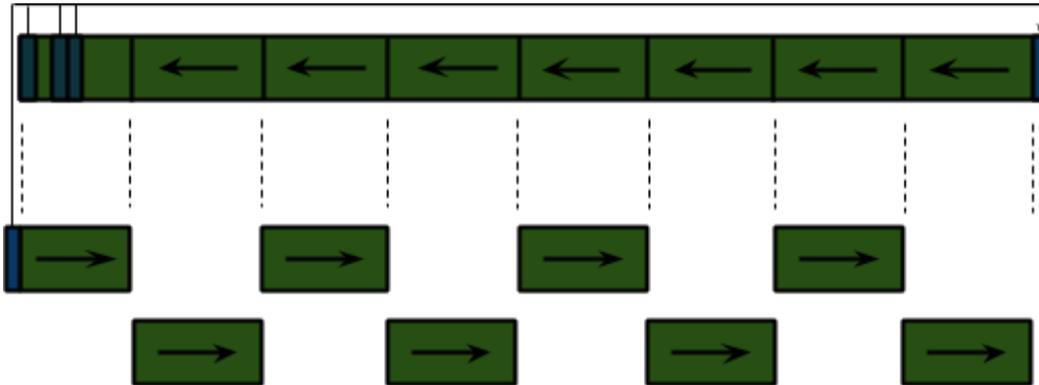
Les registres r10 et r11 composent ensemble un nombre de 64 bits.

A chaque étape, nous prenons 4 bits de ce nombre et selon leur parité (équivalent d'un XOR entre ces bits), obtenons le futur bit de poids fort de la valeur suivante ; celle-ci étant le résultat d'un décalage de 1 bit vers la droite.

Nous perdons donc la valeur du bit de poids faible, mais celui-ci participant au calcul du nouveau bit de poids fort, il est en quelque sorte réintroduit (et peut être retrouvé lors d'une opération inverse).

A chaque fois, le nouveau bit de poids faible est repris en l'état pour former un masque de 8 bits, appliqué avec un XOR sur la chaîne à décrypter pour obtenir la version en clair du message, octet par octet.

Nous savons que les 8 derniers octets en clair doivent être nuls. Nous pouvons donc en déduire la valeur de r10/r11 juste avant qu'ils n'aient été déchiffrés.



Nous reprenons pour cela les 8 derniers octets chiffrés et les plaçons, en inversant l'orientation des bits, avec un décalage de 1 bit. Le bit de poids faible pouvant être calculé.

Nous obtenons alors la valeur de 0x8195e2a78654daad.

En réappliquant l'algorithme, nous obtenons la valeur finale de 0x9c622e0dedf95d60.

Ensuite nous pouvons remonter bit par bit, en réalisant un décalage à gauche de 1 bit, et en recalculant la valeur du nouveau bit de poids faible grâce à un XOR sur les bits 0xd8000000.

Grâce au script ruby suivant, nous obtenons les valeurs originelles de r10 et r11 :

```
def parity(x)
  x = x ^ (x >> 1)
  x = (x ^ (x >> 2)) & 0x11111111
  x = x * 0x11111111
  return (x >> 28) & 1
end

r10 = 0x9c622e0d
r11 = 0xedf95d60

(0x200 * 2 * 64).times {
  r9 = parity(r10 & 0xd8000000)
  r10 = ((r10 << 1) | (r11 >> 31)) & 0xffffffff
  r11 = ((r11 << 1) | r9) & 0xffffffff
}

puts "r10 = 0x%.08x\nr11 = 0x%.08x" % [ r10, r11 ]
r10 = 0x05blad0b
r11 = 0x1ladde15
```

Ce qui nous permet de retrouver la clef attendue en remplaçant les octets dans le bon ordre :

“0BADB10515DEAD11”

Chapitre 7 - payload.bin / Reverse du “firmware”

Le fichier “payload.bin” semble être une archive au format zip, contenant 2 fichiers :

```
$ ls -l payload.bin
-rw-rw-r-- 1 peio peio 1544 mai 14 19:04 payload.bin

$ file payload.bin
payload.bin: Zip archive data, at least v2.0 to extract

$ 7z t payload.bin

7-Zip [64] 9.20 Copyright (c) 1999-2010 Igor Pavlov 2010-11-18
p7zip Version 9.20 (locale=fr_FR.UTF-8,Utf16=on,HugeFiles=on,4 CPUs)

Processing archive: payload.bin

Testing      mcu/upload.py
Testing      mcu/fw.hex

Everything is Ok

Files: 2
Size:        2570
Compressed: 1544
```

Le fichier “upload.py” nous apprend que nous sommes face à une architecture inconnue. Il reprend le fichier “fw.hex”, et une fois envoyé par une socket, affiche les données reçues.

Le fichier “fw.hex” reprend le format habituellement employé pour des firmwares. En prenant les 3 dernières lignes du fichier

Taille	Offset	Fin?	Données	Checksum
:10	01C0	00	8A0F5AE8B5D40D6CE86AA6ACC492F8F1	6F
:0C	01D0	00	72A77CE6D5A5680921D44100	87
:00	0000	01		FF

Nous constatons alors que ce firmware est un bloc binaire de 476 octets.

Nous y noterons les chaînes suivantes :

0x017C	“YeahRiscIsGood!”
0x018C	“Firmware v1.33.7 starting.\n”
0x01A8	“Halting.\n”

Nous ne connaissons rien du processeur, afin de réaliser un reverse sur ce firmware, il nous faudra donc faire des hypothèses et au besoin revenir dessus si elles n'aboutissent pas.

- Nous supposerons d'abord qu'il s'agit d'une architecture type RISC (une des chaînes de caractères présentes nous le conforte), avec 2 octets par instructions et un faible jeu d'instructions, basé sur des manipulations de registres.
- Nous pouvons raisonnablement opter sur 4 ou 5 bits (tout au plus 6) pour l'opcode, classiquement placés dans les bits de poids forts des instructions.
- Nous devrions aussi avoir au moins 8 ou 16 registres (représentés respectivement sur 3 ou 4 bits) - voire plus sinon.
- Les instructions doivent faire intervenir à minima des chargements de valeurs immédiates, des opérations entre les registres et des branchements (à priori offsets relatifs au PC).

Une fois ces hypothèses (et leurs marges) données, nous pouvons analyser le firmware, que l'on aura extrait dans un fichier "fw.bin" :

```
0x00000000: 2100111B 2001108C C0D22010 10002101      !... ..!..
0x00000010: 117C2200 120FC03C 20101000 210111B2      .|"....< ...!...
0x00000020: 22001229 C0762011 1000C0B4 C0B65A00      "...).v .....Z.
0x00000030: 21001124 200110B2 C0BE51AA C10A2100      !..$ .....Q...!.
0x00000040: 11292001 10B2C094 21001109 200110A8      .) .....!... ..
(...)
```

Nous constatons d'abord un grand nombre d'instructions commençant par '1' et '2'. Ce qui nous conforte dans le choix de 2 octets par instruction. On remarquera aussi que ces opcodes se suivent systématiquement en conservant le modèle "2x.. 1x..", le x étant identique. Cela ne nous permet pas de définir la taille des opcodes (4 à 6 bits possibles). Mais il pourrait s'agir d'une architecture big-endian si l'opcode est sur les bits de poids forts du premier octet.

Le deuxième couple est très intéressant : "2001 108C". Or nous savons qu'en 018C se trouve une des chaînes lisibles. Nous pouvons en déduire que ces opcodes permettraient de charger une valeur immédiate dans un registre, par accoup de 8 bits (instructions du type mov.h / mov.l).

Il reste donc 8 bits dans ces instructions pour représenter l'opcode et le numéro de registre. Nous avons 2 options viables : 4 / 4 (16 opcodes et 16 registres) ou 5 / 3 (resp. 32 et 8).

Si nous envisageons un opcode sur 4 bits et regroupons les instructions selon celui-ci :

```
irb> File.binread('fw.bin')[0, 0x17c].unpack('S>*').map{|n| '%.04x' % n }.group_by{|s| s[0] }.each_value{|opc| p opc }
```

Nous obtenons l'ensemble des instructions classées sur le premier quartet (opcode ?) :

```
["2100", "2001", "2010", "2101", "2200", "2010", "2101", "2200", "2011", "2100", "2001",  
"2100", "2001", "2100", "2001", "2101", "2200", "2200", "2300", "2300", "2400", "2100",  
"2201", "2300", "2410", "2500", "2600", "2700", "2300", "2300", "2427", "2500", "2700",  
"2800", "2800"]  
["111b", "108c", "1000", "117c", "120f", "1000", "11b2", "1229", "1000", "1124", "10b2",  
"1129", "10b2", "1109", "10a8", "1100", "1201", "1201", "13ff", "13ff", "1401", "1101",  
"1200", "1301", "1400", "150f", "160a", "1701", "1337", "1330", "1410", "150a", "1701",  
"1820", "1830"]  
["c0d2", "c03c", "c076", "c0b4", "c0b6", "c0be", "c10a", "c094", "c08a", "c801", "c802",  
"c803"]  
["5a00", "51aa", "5800", "5911", "5a22", "5100", "5113", "5800", "5911", "5a22", "5100",  
"5200", "5003", "5113", "5553", "5444", "5225", "5444"]  
["b084", "b3f6", "b002", "b000", "b3fc", "b3f0", "b004", "b3dc"]  
["3000", "3000", "3000", "3665", "3222", "3444", "3000", "3333", "3666", "3333"]  
["7310", "7404", "7430", "75a2", "7111", "7441", "7326", "7247", "7007", "7113"]  
["a006", "afe2", "a7dc", "a008", "a006", "a806", "a006", "a3ea", "a7de"]  
["f080", "f580", "f481", "f680", "f581", "f692", "f203", "f803", "f803"]  
["6002", "6114", "6114", "6002", "6024", "6115", "6556", "6224", "6003", "6232", "6007",  
"6357", "6007", "6882"]  
["e480", "e494", "e480", "e581", "e580", "e580", "e681", "e585", "e692", "e301", "e401",  
"e402"]  
["940a", "9214", "9443", "9214", "9445"]  
["844a", "8442", "8324"]  
["d00f", "d00f", "d00f", "d00f", "d00f", "d00f", "d00f", "d00f"]  
["4034", "4662"]
```

Les opcodes commençant par '3' sont assez remarquables : les 3 quartets de bits suivants sont souvent identiques. Cela fait penser à une instruction très fréquente : `xor rn, rn, rn`, habituellement employée pour remettre un registre à 0.

Les opcodes '5', '6', '7', '8' et 'a', peut-être '4', '9' et/ou 'b' ont aussi l'air de manipuler des quartets de bits, avec les répétitions que l'on s'attend à avoir sur un modèle

“opcode r1, r2, r3”.

En effet, ce modèle est basé sur un registre destination et 2 sources, or souvent le résultat est placé dans un registre qui était aussi présent en source.

Par ailleurs, l'opcode 'd' est assez particulier, utilisé 8 fois et toujours sous la forme “d00f”.

L'opcode 'c' donne cependant l'impression d'avoir des valeurs plus aléatoires. Il pourrait donc s'agir d'un des opcodes de branchement. On pourra remarquer 3 instructions particulières : "c801", "c802" et "c803".

Si nous ajoutons la position (offset) au paramètre de ces instructions, nous obtenons⁶ :

```
irb> File.binread('fw.bin')[0, 0x17c].unpack('S>*').each_with_index.select{|n, i| n >> 12
== 0xc }.map{|n, i| '%.04x' % ((n & 0x0fff) + i*2 + 2) }
=> ["00dc", "0054", "009c", "00e0", "00e4", "00f8", "0148", "00dc", "00dc", "08db",
"08e0", "08e5"]
```

En dehors des 3 dernières formulations déjà vues, on remarquera qu'elle restent toutes à l'intérieur du code et que la même valeur ressort 3 fois : 0x00dc. Il pourrait donc s'agir de branchement, sous forme d'appel de fonction. Pour le confirmer, nous allons regarder à quoi ressemblent les instructions qui précèdent juste celles où nous atterrissons :

```
irb> x = [0x0dc, 0x054, 0x09c, 0x0e0, 0x0e4, 0x0f8, 0x148, 0x0dc, 0x0dc]
irb> File.binread('fw.bin')[0, 0x17c].unpack('S>*').each_with_index.select{|n, i|
x.include?(i*2 + 2) }.map {|n, _| '%.04x' % n }
=> ["b084", "b002", "b3fc", "d00f", "d00f", "d00f", "d00f"]
```

Ceci nous permet à la fois de confirmer notre hypothèse, et de supposer que les opcodes 'b' seraient aussi des branchements (les opcodes 'c' étant les appels de fonctions conservant l'adresse de retour) et les opcodes 'd' seraient les retours de fonctions.

Le 'f' final pourrait laisser entendre qu'un call place l'adresse de retour dans le registre n°15.

En observant les instructions de l'opcode 'b', nous pouvons constater que les offsets sont additionné au PC (déjà incrémenté) sur un modulo de 0x0400. L'instruction "b3fc", par exemple, effectue donc un saut en arrière.

Il nous reste cependant à trouver les sauts conditionnels. L'opcode 'a' semble être un bon candidat, nous essayerons de le confirmer par la suite. Cependant, s'il s'agit bien de lui, certains bits sont encore à comprendre (peut-être selon les conditions de saut ?).

Car si nous retrouvons bien un "a3ea" (en arrière donc), nous avons aussi par exemple "a7dc" ou "a806". Nous avons donc 2 bits entre l'opcode et l'offset que nous devons comprendre.

⁶ Nous nous sommes ici basé sur une adresse relative à un PC déjà incrémenté (un "jump 0" saute sur l'instruction suivante). Nous aurions pu aussi refaire la suite de l'analyse en nous basant sur un PC non modifié ("jump 0" = boucle sans fin) si cela avait été nécessaire.

Après cette analyse, le début du code pourrait donc être :

```
0000: mov r1, 0x001b          ; v Longueur
      mov r0, 0x018c      ; "Firmware v1.33.7 starting.\n"
      call 00dc
      mov r0, 0x1000
      mov r1, 0x017c      ; "YeahRiscIsGood!"
      mov r2, 0x000f      ; ^ Longueur
      call 0054
      mov r0, 0x1000
      mov r1, 0x01b2      ; Données "aléatoires" à la suite des chaînes lisibles
      mov r2, 0x0029      ; ^ Longueur
      call 009c
      mov r0, 0x1100
      call 00e0
      call 00e4
      (? 5 ?) ra, r0, r0   ; opcode '5'
      mov r1, 0x0024
      mov r0, 0x01b2
      call 00f8
      (? 5 ?) r1, ra, ra   ; opcode '5'
      call 0148
      mov r1, 0x0029
      mov r0, 0x01b2
      call 00dc
      mov r1, 0x0009      ; v Longueur
      mov r0, 0x01a8      ; "Halting.\n"
      call 00dc
      jmp 00d8
```

On peut donc supposer que la fonction en 00dc affiche une chaîne. Or il s'y situe "c802" et "d00f". Nous pourrions donc comprendre que les "c8.." soient des appels systèmes.

Nous avons :

```
00d8: int 1 ; exit
      jmp 00d8
00dc: int 2 ; write
      ret
00e0: int 3 ; ??
      ret
```

L'opcode '5' semble permettre de copier un registre dans un autre, en mettant r0 en ra, puis en le repositionnant plus tard en r1.

Il effectue donc une copie lorsque l'instruction est sous la forme "opcode rd, rs, rs".

Il pourrait à priori s'agir de "or" ou de "and".

Analyse de la fonction 0054

```
0054: (? 5 ?) r8, r0, r0          ; 0x1000
      (? 5 ?) r9, r1, r1          ; Pointeur sur chaîne
      (? 5 ?) ra, r2, r2         ; Longueur
      xor r0, r0, r0
      mov r1, 0x0100
      mov r2, 0x0001
0064: (? 7 ?) r3, r1, r0
      (? a.00 ?) 006e
      (? f ?) r0, r8, r0
      (? 6 ?) r0, r0, r2
      jmp 0064
006e: xor r0, r0, r0
      (? 5 ?) r1, r0, r0
      mov r2, 0x0001
      mov r3, 0x00ff
007a: (? e ?) r4, r8, r0
      (? 6 ?) r1, r1, r4
      (? 9 ?) r4, r0, ra
      (? 8 ?) r4, r4, ra
      (? 7 ?) r4, r0, r4
      (? 9 ?) r4, r9, r4
      (? 6 ?) r1, r1, r4
      (? 5 ?) r1, r1, r3
      (? e ?) r4, r8, r0
      (? e ?) r5, r8, r1
      (? f ?) r5, r8, r0
      (? f ?) r4, r8, r1
      (? 6 ?) r0, r0, r2
      (? 7 ?) r4, r3, r0
      (? a.11 ?) 007a
      ret
```

L'hypothèse de l'opcode 'a' comme saut conditionnel se confirme, les offset concordent (en prenant toujours un modulo sur 0x0400). Les 2 bits restant sont indiqués en binaire.

Dans la première boucle, r0 démarre à 0, et on le combine avec r2 qui vaut 1. Il participe ensuite au test de fin de boucle.

Le registre r2 serait donc un compteur et on peut donc supposer que l'opcode '6' serve à l'addition.

A noter une étrange construction :

```
(? e ?) r4, r8, r0
(? e ?) r5, r8, r1
(? f ?) r5, r8, r0
(? f ?) r4, r8, r1
```

Analyse de la fonction 009c

```
009c: jmp 009e
009e: (? 5 ?) r8, r0, r0
      (? 5 ?) r9, r1, r1
      (? 5 ?) ra, r2, r2
      xor r0, r0, r0
      (? 5 ?) r1, r0, r0
      (? 5 ?) r2, r0, r0
      mov r3, 0x00ff
      mov r4, 0x0001
00b2: add r0, r2, r4
      (? 5 ?) r0, r0, r3
      (? e ?) r5, r8, r0
      add r1, r1, r5
      (? 5 ?) r1, r1, r3
      (? e ?) r5, r8, r0
      (? e ?) r6, r8, r1
      (? f ?) r6, r8, r0
      (? f ?) r5, r8, r1
      add r5, r5, r6
      (? 5 ?) r5, r5, r3
      (? e ?) r5, r8, r5
      (? e ?) r6, r9, r2
      xor r6, r6, r5
      (? f ?) r6, r9, r2
      add r2, r2, r4
      (? 7 ?) r5, ra, r2
      jcond.01 00b2
      ret
```

Dans la boucle 00b2, r4 conserve la valeur de 1, tandis que r2 initialisé à 0 et incrémenté à chaque tour. La construction

```
add r0, r2, r4
(? 5 ?) r0, r0, r3 ; r0 &= 0xff (?)
```

Laisserait plutôt entendre que l’opcode ‘5’ soit un “and” (un “or” n’aurait pas trop de sens ici).

On remarquera que tout comme la fonction précédente, les sauts conditionnels sont précédés de l’opcode ‘7’. Or l’instruction la plus utilisée dans ce contexte serait un “sub”, permettant la comparaison. On voit d’ailleurs que le résultat est placé dans un registre (ici : r5) qui n’est pas utilisé par la suite, le rôle est bien de faire une comparaison, mais au travers d’une instruction qui effectue réellement une action de soustraction (il ne s’agirait donc pas juste d’un “test”).

On retrouve la même construction que précédemment :

```
(? e ?) r5, r8, r0
(? e ?) r6, r8, r1
(? f ?) r6, r8, r0
(? f ?) r5, r8, r1
```

Dans les deux cas nous faisons intervenir un registre r8 dont le contenu est 0x1000, accompagné d’un compteur. Nous pourrions donc être en présence d’opcodes de manipulation de mémoire.

Analyse des fonctions 00e4 et 00f8

```
00e4: mov r1, 0x0001
      mov r2, 0x0100
      (? e ?) r3, r0, r1
      sub r1, r1, r1
      (? e ?) r4, r0, r1
      (? 8 ?) r4, r4, r2
      (? 4 ?) r0, r3, r4
      ret

00f8: xor r2, r2, r2          ; r2 est positionné à 0 et n'est pas modifié dans la
fonction
      mov r3, 0x0001
00fe: xor r4, r4, r4
      (? e ?) r4, r0, r2
      and r4, r4, r4
      jcond.00 010e
      sub r4, r4, r1
      jcond.00 0110
      add r0, r0, r3
      jmp 00fe
010e: xor r0, r0, r0
0110: ret
```

Si nos hypothèses sont bonnes, la fonction 00f8 est quasiment désassemblée.

Ici le registre r4 est mis à 0, utilisé dans l'opcode 'e' avant d'être testé par un "and" (afin de savoir s'il a une valeur nulle sans en changer le contenu). C'est donc que cet opcode 'e' modifie le registre r4, et si nous sommes bien en présence d'une manipulation mémoire, il s'agit d'une lecture ("ld").

En considérant que "jcond.00" effectue un saut si le résultat de l'instruction précédente était nul ("jz"), la boucle va parcourir la chaîne pointée par r0, tant que les octets sont différents de '0' et de r1. Si nous rencontrons un '0', la fonction retourne 0. Si c'est un octet ayant la même valeur que r1 qui est rencontré, on retourne son index.

Cette fonction est l'équivalent d'un "strchr".

L'opcode 'e' effectue la somme de ses 2 derniers paramètres et retourne le contenu à cette adresse dans le registre donné en premier paramètre.

La fonction 00e4 charge donc 2 octets consécutifs à l'adresse r0 et r0 + 1. Le premier subit une opération avec le nombre 0x0100 avant d'être combiné avec le second pour donner la valeur de retour.

Cela ressemble fortement à la lecture d'un mot de 2 octets, en lisant d'abord celui de poids fort, multiplié par 0x100, puis on effectue un OR avec celui de poids faible.

L'opcode '8' serait donc la multiplication et l'opcode '4' l'opération "or".

Analyse de la fonction 0112 (non utilisée)

```
    mov r4, 0x1000
    mov r5, 0x000f
    mov r6, 0x000a
    mov r7, 0x0001
0122: (? 9 ?) r2, r1, r4
    and r2, r2, r5
    sub r3, r2, r6
    jcond.10 0130
    mov r3, 0x0037
    jmp 0134
0130: mov r3, 0x0030
0134: add r2, r3, r2
    xor r3, r3, r3
    (? f ?) r2, r0, r3
    add r0, r0, r7
    sub r2, r4, r7
    jz 0146
    add r3, r5, r7
    (? 9 ?) r4, r4, r3
    jmp 0122
0146: ret
```

A l'intérieur de la boucle, nous comparons (r2 & 0xf) avec 10, et selon le résultat prenons comme repère dans r3 : 0x30 ou 0x37. Ce repère étant additionné avec la valeur (r2 & 0xf). Cela ressemble à une conversion hexadécimale d'un quartet de bit en ASCII.

Le résultat étant alors stocké dans la chaîne pointée par r0. Pour cela, l'opcode 'f' nous permet d'écrire la valeur d'un registre en mémoire ("st").

La comparaison est donc '< 0', et nous pouvons renommer "jcond.10" en "js".

Aussi le compteur de boucle démarre à 0x1000, combiné avec 0x10 (somme de 0xf et 1) à chaque tour pour passer au quartet suivant. L'opcode '9' réalise donc la division.

L'ensemble des instructions rencontrées peuvent maintenant être désassemblées⁷.

1xxx et 2xxx : (mov.l / mov.h) mov

3xxx, 4xxx, 5xxx : xor, or, and

6xxx, 7xxx, 8xxx, 9xxx : add, sub, mul, div

a(n)xx, selon la valeur de (n) : [0-3] jz, [4-7] jnz, [8-b] js, [c-f] jns

bxxx : jmp

c(n)xx, pour n < 4 : call, pour n = 8 : int

d00(n), pour n = f : ret, sinon on peut supposer "br r" ?

exxx, fxxx : ld, st

⁷ Annexe E : Outils MCU

Nouvelle analyse de la fonction 0054

```
0054:  and r8, r0, r0
      and r9, r1, r1
      and ra, r2, r2
      xor r0, r0, r0
      mov r1, 0x0100
      mov r2, 0x0001
0064:  sub r3, r1, r0
      jz 006e
      st r0, r8, r0
      add r0, r0, r2
      jmp 0064
006e:  xor r0, r0, r0
      and r1, r0, r0
      mov r2, 0x0001
      mov r3, 0x00ff
007a:  ld r4, r8, r0
      add r1, r1, r4
      div r4, r0, ra
      mul r4, r4, ra
      sub r4, r0, r4
      ld r4, r9, r4
      add r1, r1, r4
      and r1, r1, r3
      ld r4, r8, r0
      ld r5, r8, r1
      st r5, r8, r0
      st r4, r8, r1
      add r0, r0, r2
      sub r4, r3, r0
      jns 007a
      ret
```

Nous y voyons maintenant plus clair. Cependant la construction suivante :

```
div r4, r0, ra
mul r4, r4, ra
sub r4, r0, r4
```

permet de calculer un modulo (le jeu d'instruction ne l'implémentant pas).

La fonction initialise un tableau de 256 octets à l'adresse en premier paramètre (0x1000), en le remplissant de 0 à 255.

Ensuite nous reprenons chacun de ces octets, et les combinons avec la chaîne fournie en deuxième paramètre ("YeahRisclsGood!") en cyclant dès qu'on arrive à la fin de celle-ci.

Cette combinaison consiste à réaliser des échanges de valeurs entre l'index en cours et celui calculé à partir de cette chaîne/clef.

Nous retrouvons l'algorithme d'initialisation key-schedule de RC4.

Nouvelle analyse de la fonction 009c

```
009c:  jmp 009e
009e:  and r8, r0, r0
      and r9, r1, r1
      and ra, r2, r2
      xor r0, r0, r0
      and r1, r0, r0
      and r2, r0, r0
      mov r3, 0x00ff
      mov r4, 0x0001
00b2:  add r0, r2, r4
      and r0, r0, r3
      ld r5, r8, r0
      add r1, r1, r5
      and r1, r1, r3
      ld r5, r8, r0
      ld r6, r8, r1
      st r6, r8, r0
      st r5, r8, r1
      add r5, r5, r6
      and r5, r5, r3
      ld r5, r8, r5
      ld r6, r9, r2
      xor r6, r6, r5
      st r6, r9, r2
      add r2, r2, r4
      sub r5, ra, r2
      jnz 00b2
      ret
```

Nous retrouvons ici sans surprise maintenant le code de génération du flot pseudo-aléatoire RC4 et chaque octet est XOR'é avec la chaîne initialement cryptée fournie en paramètre.

Si nous appliquons l'algorithme RC4 sur le buffer en 0x01b2 en initialisant la clef avec la chaîne "YeahRisclsGood!", nous obtenons la chaîne
"Execution completed in \$\$\$\$ CPU cycles."

Le programme principal recherche ensuite la position du symbole '\$'.
Nous exécutons le dernier appel système (int 3), en fournissant une adresse à remplir, et nous remplaçons les '\$' par le contenu fourni par le kernel à cette adresse, converti en décimal. Cette chaîne est affichée, suivie par "Halting." et le programme s'arrête.

Chapitre 8 - Dumper la “Secret memory area”

Une fois le processeur mieux maîtrisé, il ne sert à rien de passer près de 2 semaines à retourner le firmware fourni dans tous les sens, afin de repérer un hypothétique semblant d’adresse email lors des calculs RC4 par exemple.

La clef se situe dans le fichier “upload.py”, qui ne s’arrête pas à juste pousser le firmware sur une socket TCP, il le fait sur une IP publique. L’architecture mystérieuse existe donc bel et bien (même si elle est certainement virtuelle), et nous pouvons nous y connecter :

```
$ cat fw.hex | nc 178.33.105.197 10101
System reset.
Firmware v1.33.7 starting.
Execution completed in 8339 CPU cycles.
Halting.
```

L’objectif va donc maintenant être de réaliser des programmes (ou “firmwares”), et les faire exécuter par ce MCU, et de réussir à obtenir le contenu de la mémoire dite “Secret memory area”.

Il va donc falloir réaliser d’une part un assembleur⁸, et d’autre part générer le fichier firmware. Nous avons tous les éléments sauf le calcul du checksum.

Après quelques essais, on peut rapidement constater que le checksum est obtenu, de manière à ce que la somme de tous octets de la ligne (sur un masque de 0xff) fasse 0.

Pour la ligne “:0C01D00072A77CE6D5A5680921D44100”, nous obtenons :

```
irb> b = '0C01D00072A77CE6D5A5680921D44100'
irb> '%.02X' % ( - [b].pack('H*').each_byte.inject(0x0) {|x, s| (s + x) % 256 } & 0xff )
=> "87"
```

Qui est bien la valeur retrouvé dans le fichier “fw.hex”.

Nous pouvons dumper l’ensemble de la mémoire utilisateur afin de voir si nous pouvons avoir des indices supplémentaires, mais nous ne verrons rien d’autres que des ‘0’.

Il faudra donc utiliser les quelques appels systèmes à notre disposition.

⁸ Annexe E : Outils MCU

Nous ne pouvons pas non plus dumper directement la "Secret memory area" :

```
$ cat blutch.asm
start: mov r0, 0xf000
      mov r1, 768
      int 2

fin:   int 1
      jmp fin
```

```
$ ruby asm.rb blutch.asm | nc 178.33.105.197 10101
System reset.
[ERROR] Printing at unallowed address. CPU halted.
```

Par contre, les autres parties privilégiées sont accessibles par l'appel système write :

```
$ cat blutch.asm
start: mov r0, 0xfc00
      mov r1, 1024
      int 2

fin:   int 1
      jmp fin
```

```
$ ruby asm.rb blutch.asm | nc 178.33.105.197 10101 | hd -vs 14
0000000e 0a 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
0000001e 00 00 03 ef 00 00 00 00 00 00 00 00 00 00 00 |.....|
0000002e fc 00 04 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
0000003e 00 00 00 00 00 00 00 00 00 00 ef fe 00 00 00 00 |.....|
0000004e 00 0a 00 03 00 00 00 00 00 00 00 00 00 00 00 |.....|
0000005e 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
0000006e 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
0000007e 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
0000008e 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
0000009e 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000ae 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000be 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000ce 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000de 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000ee 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000fe 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
0000010e 50 00 a0 6c 21 00 11 03 72 10 a8 12 22 00 12 02 |P..l!...r..."...|
0000011e 81 02 71 12 20 f0 10 00 60 01 c0 94 d0 00 21 00 |..q. ....`.....!|
0000012e 11 2b 20 fe 10 5a c0 be 30 00 21 fc 11 10 22 00 |.+ ..Z..0!..."|
0000013e 12 01 f2 10 b3 f2 20 fc 10 22 c0 74 55 00 20 fc |.....".tU. .|
0000014e 10 20 c0 6c 51 55 c0 9e d8 00 20 fc 10 20 c0 60 |. .lQU.... ..`|
0000015e 26 fc 16 12 21 00 11 01 34 44 e5 61 e2 64 e3 64 |&...!...4D.a.d.d|
0000016e 73 32 a7 f6 23 01 13 00 82 23 41 25 c0 56 d8 00 |s2..#....#A%.V..|
0000017e 21 00 11 0e 20 fe 10 86 c0 6c 24 00 14 02 21 fd |!... ..l$....!|
0000018e 11 28 20 f0 10 00 c0 3c 60 04 21 fd 11 36 c0 34 |.( ....<`.!...6.4|
0000019e 60 04 21 fd 11 4a c0 2c 20 fc 10 20 31 11 22 00 |`.!..J., .. 1."|
000001ae 12 36 c0 32 20 fc 10 3a 21 ef 11 fe c0 16 d8 00 |.6.2 ...:!......|
000001be 21 00 11 01 22 01 12 00 e3 01 71 11 e4 01 84 42 |!..."...q...B|
000001ce 40 34 d0 0f 22 00 12 01 23 01 13 00 f1 02 72 22 |@4..."...#.....r|
000001de 91 13 f1 02 d0 0f 23 00 13 01 52 22 a0 06 72 23 |.....#...R"..r#|
000001ee f1 02 b3 f2 d0 0f 5e 00 2d fc 1d 00 2c f0 1c 00 |.....^-...;...|
000001fe 38 88 59 88 2a 00 1a 01 3b bb 51 11 a0 1a 69 e8 |8.Y.*...;Q...i|
0000020e 79 9c a8 08 69 e8 79 9d ac 02 b0 0e 39 99 e9 e8 |y...i.y.....9...|
```

```

0000021e f9 db 68 8a 71 1a b3 e2 d0 0f 21 00 11 33 20 fe |..h.q.....!...3 .|
0000022e 10 26 c3 c2 b3 02 5b 45 52 52 4f 52 5d 20 50 72 |.&....[ERROR] Pr|
0000023e 69 6e 74 69 6e 67 20 61 74 20 75 6e 61 6c 6c 6f |inting at unallo|
0000024e 77 65 64 20 61 64 64 72 65 73 73 2e 20 43 50 55 |wed address. CPU|
0000025e 20 68 61 6c 74 65 64 2e 0a 00 5b 45 52 52 4f 52 | halted...[ERROR|
0000026e 5d 20 55 6e 64 65 66 69 6e 65 64 20 73 79 73 74 |] Undefined syst|
0000027e 65 6d 20 63 61 6c 6c 2e 20 43 50 55 20 68 61 6c |em call. CPU hal|
0000028e 74 65 64 2e 0a 00 53 79 73 74 65 6d 20 72 65 73 |ted...System res|
0000029e 65 74 2e 0a 00 00 00 00 00 00 00 00 00 00 00 00 |et.....|
000002ae 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000002be 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000002ce 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000002de 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
(...)

```

Attention, dans le dump précédent, les offsets sont décallés (il s'agit des offsets relatifs à ce que voit "hd").

Le début du dump correspond à l'adresse 0xfc00 (HW Registers)

La ROM kernel débutant en 0xfd00 commence à la ligne 0000010e

De nouveau nous pouvons noter quelques chaînes lisibles :

0xfe26	"[ERROR] Printing at unallowed address. CPU halted.\n"
0xfe5a	"[ERROR] Undefined system call. CPU halted.\n"
0xfe86	"System reset.\n"

Si nous désassemblons la ROM, nous pouvons mieux comprendre son fonctionnement et celui de l'architecture qui l'accueille.

```

fd00:  and r0, r0, r0
      jz fd70
      mov r1, 0x0003
      sub r2, r1, r0
      js fd1e
      mov r2, 0x0002
      mul r1, r0, r2
      sub r1, r1, r2
      mov r0, 0xf000
      add r0, r0, r1
      call fdb0
      br r0

fd1e:  mov r1, 0x002b
      mov r0, 0xfe5a
      call fde6

```

Ce code en début de ROM semble répondre à un appel système, en prenant le numéro d'interruption dans le registre r0. Si c'est bien le cas, lors d'une interruption, les registres tels qu'ils étaient en mode User sont peut-être sauvegardés (dans la zone HW Registers ?), le n° d'int est placé en r0, le mode bascule en privilégié et l'exécution reprendrait à l'offset 0xfd00.

Lors d'une interruption, nous voyons que si r0 est bien compris entre 0 et 3, nous appelons la fonction indiquée dans le tableau de pointeurs située à l'adresse 0xf000 (dans la zone secrète donc).

Par ailleurs nous retrouvons la fonction write en interne du kernel à l'adresse 0xfde6.

```
fde6:  and re, r0, r0
      mov rd, 0xfc00
      mov rc, 0xf000
      xor r8, r8, r8
      and r9, r8, r8
      mov ra, 0x0001
      xor rb, rb, rb
fdfa:  and r1, r1, r1
      jz fe18                ; Fin de la chaîne à afficher
      add r9, re, r8
      sub r9, r9, rc
      js fe0c                ; offset < 0xf000 => ok
      add r9, re, r8
      sub r9, r9, rd
      jns fe0c               ; offset >= 0xfc00 => ok
      jmp fe1a
fe0c:  xor r9, r9, r9
      ld r9, re, r8
      st r9, rd, rb
      add r8, r8, ra
      sub r1, r1, ra
      jmp fdfa
fe18:  ret

fe1a:  mov r1, 0x0033
      mov r0, 0xfe26
      call fde6              ; Affichage du message d'erreur
      jmp fd28
```

Nous voyons que la fonction lit octet par octet la chaîne à afficher, et vérifie si l'offset de cet octet n'est pas dans la zone secrète. Dans le cas positif, elle l'écrit en 0xfc00, qui doit donc représenter le périphérique de sortie. Dans le cas négatif, une erreur d'accès est affichée.

Si nous arrivons à exécuter du code en mode privilégié, nous pouvons donc réécrire cette fonction sans le check "de sécurité".

En cas de “int 0”, nous avons vu que la fonction appelée était 0xfd70.

```
fd70: mov r1, 0x000e
      mov r0, 0xfe86           ; "System reset."
      call fde6
      mov r4, 0x0002
      mov r1, 0xfd28
      mov r0, 0xf000
      call fdc4               ; [0xf000] = 0xfd28
      add r0, r0, r4
      mov r1, 0xfd36
      call fdc4               ; [0xf002] = 0xfd36
      add r0, r0, r4
      mov r1, 0xfd4a
      call fdc4               ; [0xf004] = 0xfd4a
      mov r0, 0xfc20
      xor r1, r1, r1
      mov r2, 0x0036
      call fdd6               ; memset(0xfc20, 0, 0x36)
      mov r0, 0xfc3a
      mov r1, 0xeffe
      call fdc4               ; [0xfc3a] = 0xeffe
      reti

fdc4: mov r2, 0x0001           ; Cette fonction écrit r1 (2 octets) à l'adresse r0
      mov r3, 0x0100
      st r1, r0, r2
      sub r2, r2, r2
      div r1, r1, r3
      st r1, r0, r2
      ret

fdd6: mov r3, 0x0001           ; memset(r0, r1, r2)
      and r2, r2, r2
      jz fde4
      sub r2, r2, r3
      st r1, r0, r2
      jmp fdd6
fde4: ret
```

Nous rencontrons le nouvel opcode “d800”, pendant de “c8xx”, nous le nommerons “reti”

On obtient les adresses du tableau d’interruptions : 0xfd28, 0xfd36 et 0xfd4a.

Il y a aussi l’initialisation de registres HW : de 0xfc20 à 0xfc56 positionnés à 0, en dehors de 0xfc3a qui prend la valeur 0xeffe. Or nous avons vu par le dump des registres au démarrage que tout était à 0 à l’exception du registre r13 qui avait la valeur 0xeffe (peut-être pour pouvoir se servir de lui comme d’un pointeur de pile).

Les valeurs des registres en mode User sont donc peut-être situés à cette position.

Afin d’exécuter du code en mode privilégié, nous pouvons essayer de modifier le tableau d’interruption, et le faire pointer sur une fonction à nous en zone mémoire basse. Tant que le processeur n’a pas exécuté de “reti”, il devrait garder le mode, même si la zone mémoire n’est plus dans la ROM. Nous allons vérifier cette hypothèse.

D'abord nous allons regarder le seul appel système qui réalise une écriture, mais si la valeur écrite n'est pas entièrement maîtrisée : int 3.

```
fd4a:  mov r0, 0xfc20
      call fdb0
      mov r6, 0xfc12
      mov r1, 0x0001
      xor r4, r4, r4
fd5a:  ld r5, r6, r1
      ld r2, r6, r4
      ld r3, r6, r4
      sub r3, r3, r2
      jnz fd5a
      mov r3, 0x0100
      mul r2, r2, r3
      or r1, r2, r5
      call fdc4
      reti
```

L'adresse où écrire le nombre de cycles est récupérée (depuis 0xfc20 => r0 en mode User). Ensuite, le mot à l'adresse 0xfc12 est lu, à condition qu'il n'ait pas changé de valeur sur 2 lectures consécutives de l'octet de poids fort (et éviter une incohérence).

La valeur est recomposée en r1 et écrite à l'adresse donnée par la fonction 0xdc4 vue plus haut. Il n'y a pas de vérification sur l'adresse, nous pouvons donc modifier n'importe quelle mot en mémoire (en dehors peut-être de la ROM elle-même).

Essayons de voir quelle est la première valeur générée :

```
$ cat blutch.asm
start:  mov r0, 0x1000
      int 3
      mov r1, 2
      int 2

fin:    int 1
      jmp fin

$ ruby asm.rb blutch.asm | nc 178.33.105.197 10101 | hd -vs 14
0000000e  07 c0                                     |..|
00000010
```

La valeur 0x07c0 est tout à fait satisfaisante. Nous pouvons donc l'écrire à la place du pointeur correspondant à "int 3", et un appel suivant devrait donc se rendre à cette adresse.

Nous pouvons provoquer un Invalid instruction pour vérifier que nous sommes bien toujours en Mode kernel :

```
$ cat blutch.asm
start: mov r0, 0xf004
      int 3
      int 3

fin:   int 1
      jmp fin

$ ruby asm.rb blutch.asm | nc 178.33.105.197 10101
System reset.
-- Exception occurred at 07C0: Invalid instruction.
r0:07C0   r1:0000   r2:0100   r3:00C0
r4:0700   r5:0000   r6:0000   r7:0000
r8:0000   r9:0000   r10:0000  r11:0000
r12:0000  r13:EF FE  r14:0000  r15:FD1C
pc:07C0 fault_addr:0000 [S:0 Z:0] Mode:kernel
CLOSING: Invalid instruction.
```

Tout se déroule correctement. Nous avons bien le PC en 0x07c0 et le mode est privilégié.

Nous n'avons donc plus qu'à réaliser un firmware, qui, à l'adresse 0x7c0 contienne le code nécessaire pour dumper la zone secrète :

```
$ cat blutch.asm
start: mov r0, 0xf004
      int 3
      int 3

fin:   int 1
      jmp fin

foo:   dw 0x0000
      (...)
      dw 0x0000
      dw 0x0000
      dw 0x0000
      dw 0x0000
      dw 0x0000
      dw 0x0000
```

```
_7c0: mov r0, 0xf006
      mov r1, 0x0bfa
      call write
      iret

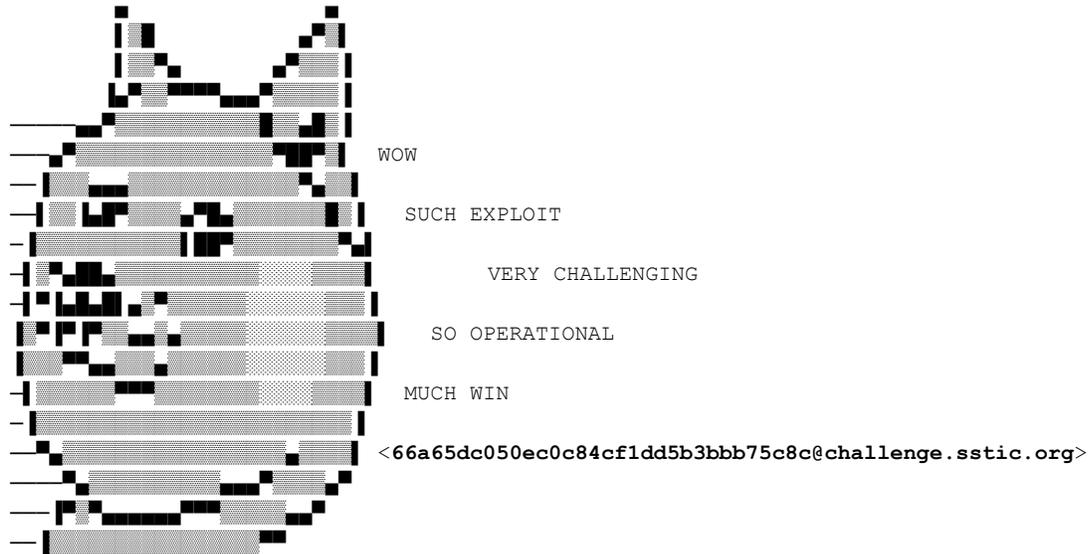
write: and re, r0, r0
      mov rd, 0x7e00
      xor r8, r8, r8
      and r9, r8, r8
      mov ra, 0x0001

b1:   and r1, r1, r1
      jz b2
      add r9, re, r8

      xor r9, r9, r9
      ld r9, re, r8
      st r9, rd, rd      ; 0x7e00 + 0x7e00 = 0xfc00
      add r8, r8, ra
      sub r1, r1, ra
      jmp b1

b2:   ret
```

```
$ ruby asm.rb blutch.asm | nc 178.33.105.197 10101
System reset.
```



Fin.

Handwritten notes on a grid background. On the left, there is a list of numbers from 1 to 20, each followed by a multiplication sign and a number (e.g., 1 x 1, 2 x 2, ..., 20 x 20). In the center, there is a diagram of a computer monitor with a keyboard below it. To the right of the monitor, there are some handwritten notes and a small table. At the bottom, there are several small diagrams or flowcharts with circles and arrows.

Handwritten notes on a grid background. At the top, there are mathematical expressions involving x_1 and x_2 . Below this, there is a large, complex flowchart with many nodes (circles) and arrows, representing a sequence of operations or a search process. To the right of the flowchart, there are several lines of mathematical formulas and text, including a recursive definition of a function $f(x_1, x_2)$.

Handwritten notes on a grid background. At the top, there is a long binary string: 0 1 1 1 0 0 1 1 0 0 0 1 0 0 0 0. Below this, there are several lines of text and mathematical expressions, including "nr: do of", "5: or", "3: and", and "12: or". There are also some small diagrams and arrows. On the right side, there is a list of numbers and their corresponding values or labels, such as "2200 1173", "2096", "2011 1030", etc. At the bottom, there are more mathematical expressions and a small table.

Et beaucoup de papiers griffonnés...