Rétroconception et débogage d'un baseband Qualcomm

Guillaume Delugré
guillaume(@)security-labs.org

Sogeti ESEC R&D

Résumé Les basebands sont omniprésents dans nos vies, et pourtant le fonctionnement de ces systèmes fermés reste un sujet très peu abordé. Quelques rares travaux ont mis en lumière des vulnérabilités dans l'implémentation des piles protocolaires téléphoniques sur différents modèles de baseband (notamment les travaux de Ralf-Philipp Weinmann présentés au 27C3 et Hack.lu). Cependant, aucune présentation ne s'est vraiment penchée sur l'analyse d'un système d'exploitation pour baseband. Cette présentation aspire à combler ce manque dans la littérature.

A partir d'une simple clé 3G équipée d'un baseband Qualcomm, il sera présenté :
- comment extraire une image de la mémoire du système
- l'architecture du micro-noyau temps réel propriétaire de Qualcomm
- comment développer un débogueur pour analyser les tâches s'exécutant sur le système

1 Introduction

1.1 Contexte et état de l'art

Les systèmes en bande de base (abrégé en *basebands* dans le reste de cet article) sont omniprésents dans nos vies. Et pour cause : tout téléphone en est équipé. Pourtant à l'heure de l'écriture de cette article, beaucoup de gens, y compris parmi les technophiles, ignorent ce qu'est un baseband.

Le processeur de bande de base est la puce d'un téléphone gérant la communication radio avec le réseau de l'opérateur téléphonique. Par extension, le baseband désigne aussi le système d'exploitation embarqué sur cette puce. Il est notamment en charge de la gestion des piles protocolaires sur le réseau cellulaire.

Historiquement, le milieu de la téléphonie a toujours été très fermé. En particulier les téléphones portables ont durant longtemps été considérés comme des boîtes noires avec lesquelles l'interaction restait très limitée

(à cette époque là, les téléphones portables servaient seulement à téléphoner...). Cette vision a cependant beaucoup changé avec l'arrivée des smartphones il y a plusieurs années. La sécurité des téléphones portables est devenue depuis un vrai sujet d'actualité.

On peut néanmoins constater l'occultation complète de la partie baseband dans le modèle de sécurité des téléphones portables. Les basebands sont à l'heure actuelle des systèmes fermés et mal connus (la seule implémentation libre étant le projet OsmocomBB[4,19]). Ils sont pourtant une cible potentiellement intéressante pour un attaquant : les piles protocolaires représentent des millions de lignes de code historiques. Les basebands sont des systèmes complexes et peu audités.

Les travaux de recherche existants sur les basebands s'intéressent principalement :

- aux attaques sur la cryptographie[14,15].
- aux attaques à distance dans l'implémentation des protocoles cellulaires[17,11,18,8].
- aux attaques locales à des fins de SIM-unlocking[10,9].
- à l'étude des protocoles pour le développement de systèmes alternatifs libres[4,20,19].

Peu d'études se sont penchées sur l'analyse du système d'un baseband propriétaire. Cette connaissance est pourtant indispensable pour quiconque serait curieux du fonctionnement de tels systèmes ou désirerait rechercher et exploiter des vulnérabilités.

Cette présentation s'intéressera donc à l'étude d'un baseband Qualcomm présent sur la série des clés USB 3G Icon 225. Il sera présenté comment extraire une image de la mémoire vive du système, l'architecture du noyau temps-réel de Qualcomm, ainsi qu'une preuve de concept de débogueur permettant d'analyser des tâches du système dynamiquement avec GDB.

2 Analyse d'une clé USB 3G : Option Icon 225

Une clé USB 3G ni plus ni moins qu'un téléphone dépourvu d'écran, de microphone et de haut-parleurs. La clé se comporte comme un modem 3G auprès de l'ordinateur hôte, en lui offrant un port série de commandes AT émulé sur le bus USB.

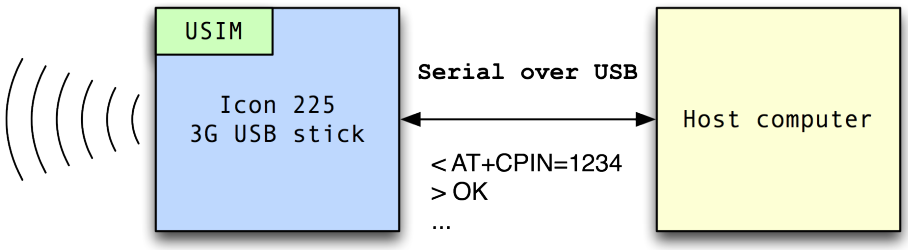

Figure 1. Fonctionnement normal d'un modem cellulaire

Nous nous intéresserons ici à la série des clés Option Icon 225 (ce modèle date de 2008). Ces clés sont équipées d'un baseband Qualcomm MSM6280, tournant sur un processeur ARMv5TEJ. Le branchement de cette clé sur un port USB déclenche la création de 3 ports série :

- Un port de communication pour les commandes AT
- Un port pour faire transiter les données une fois la communication établie sur le réseau de l'opérateur
- *Un port de diagnostic propre aux basebands Qualcomm*

Le protocole utilisé par ce port de diagnostic n'est pas officiellement documenté, mais il a cependant été en partie déjà étudié et implémenté dans le projet Gnome ModemManager.

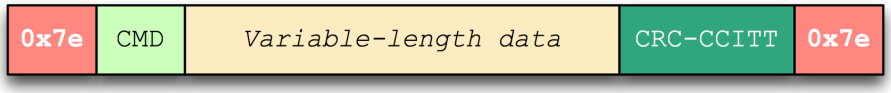

Figure 2. Format d'un paquet du protocole de diagnostic Qualcomm

Deux commandes disponibles sur ce port de diagnostic sont particulièrement intéressantes : l'une et l'autre permettant respectivement de lire et écrire la mémoire à une adresse arbitraire.

Il est donc possible très simplement d'extraire une image de la mémoire du système en cours d'exécution. Ceci permet l'inspection du tas du système, ainsi que l'état des piles de chaque tâche. Cette approche apporte beaucoup plus d'informations qu'une analyse statique sur une image de mise-à-jour du système.

Le chargeur de démarrage peut être extrait à l'adresse `0xffff0000`. La totalité du système est accessible depuis l'adresse 0 sur une longueur de 32 Mo.

Une étude du chargeur de démarrage nous indique ensuite que le point d'entrée du système se situe à l'adresse `0x80000`. En analysant la table des interruptions (disponible à l'adresse 0) et le code de démarrage du système, il est possible de rapidement remonter jusqu'au code du micro-noyau et de localiser les structures de l'ordonnanceur.

3 REX, le noyau temps-réel de Qualcomm

Le code exécuté sur le baseband est un système d'exploitation propriétaire de Qualcomm, nommé AMSS [1]. Ce système utilise un noyau temps-réel, non-documenté, nommé REX [2].

Par rétroconception, il est possible de reconstituer la plupart des primitives de ce noyau, à savoir :

- L'ordonnanceur temps-réel
- Les mécanismes de communication entre les tâches du système
- Les mécanismes d'appels APC/DPC
- Les *timers*
- La structure du tas

3.1 Démarrage du système

Dans le cas des systèmes Qualcomm, le premier chargeur de démarrage (PBL [3]) est situé en ROM et est chargé à l'adresse `0xffff0000`. Le code d'un deuxième chargeur de démarrage (QCSBL [4]) est lu depuis la

1. *Advanced Mobile Subscriber System*
2. *Real-time EXecutive*
3. *Primary Bootloader*
4. *Qualcomm Secondary Bootloader*

NVRAM et exécuté.

Sur un téléphone, ce chargeur donne normalement la main au chargeur de démarrage du constructeur (OEMSBL [5]), s'exécutant sur le processeur applicatif. Dans le cas présent, QCSBL charge et exécute directement la routine de démarrage de AMSS. Qualcomm offre la possibilité d'établir une chaîne de confiance entre chaque chargeur de démarrage. Cependant les différents chargeurs ne sont pas cryptographiquement signés sur ce modèle de clé 3G.

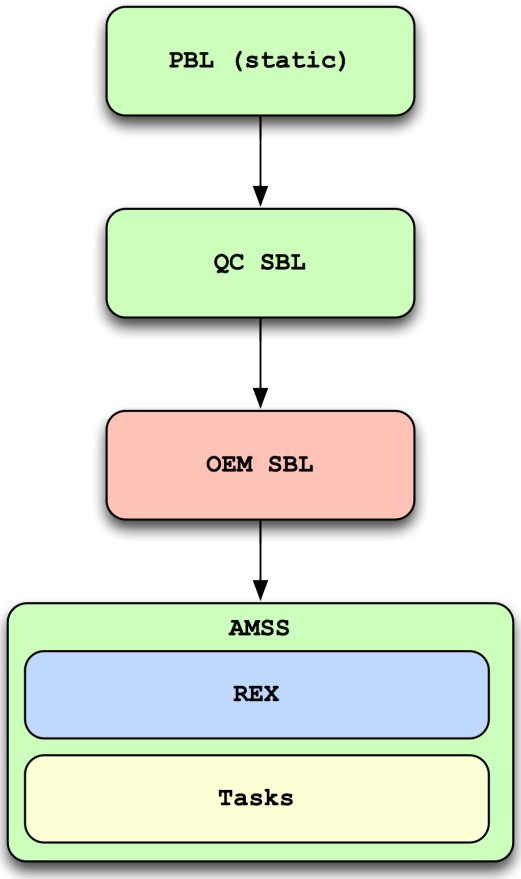

Figure 3. Démarrage du système

5. *OEM Secondary Bootloader*

Le noyau implémente des processus légers, appelés tâches, partageant tous le même espace d'adressage. La MMU est activée lors du démarrage du système, les adresses virtuelles étant égales aux adresses physiques. Les premiers 13 Mo de la mémoire sont protégés en lecture seule. Il faut noter que ARMv5 ne dispose pas du bit XN, toute la mémoire est donc exécutable.

Trois tâches sont initialement lancées au démarrage d'AMSS :
- La tâche IDLE
- La tâche DPC
- La tâche MAIN, chargée de lancer toutes les autres tâches du système

Le système démarré compte au final 69 tâches. Ces tâches peuvent être dédiées :
- A la gestion du matériel (DSP, USB, carte USIM, ...)
- A la gestion des piles réseaux (GSM, SMS, RRC, LLC, ...)
- A des fins diverses et variées (tâche de diagnostic, générateur aléatoire cryptographique...)

La plupart des tâches du système rendent compte de leur activité à une tâche *watchdog*. L'arrêt de l'une des tâches (suite à un dysfonctionnement ou un point d'arrêt par exemple) sera détecté par la tâche DOG qui redémarrera le système. Il convient donc de stopper au préalable son exécution lors d'une analyse du système.

Le nom de la tâche IDLE peut induire en erreur. Cette tâche, qui possède la plus faible priorité, n'est en réalité jamais ordonnancée. Son seul rôle est de sauvegarder le temps passé dans les routines d'interruption. La tâche ordonnancée en dernier recours est en réalité la tâche SLEEP.

3.2 Primitives du système

Ordonnancement et communication inter-tâches L'ordonnanceur de REX est de type préemptif avec priorités fixes. Lorsqu'une tâche se met en sommeil, son contexte d'exécution est sauvegardé sur sa pile. Le pointeur de pile courant est alors remplacé par celui de la prochaine tâche à exécuter, et le contexte de cette nouvelle tâche est chargé depuis la pile. Chaque tâche peut se mettre en sommeil dans l'attente d'un ou plusieurs signaux. La tâche ne peut alors être réveillée que par l'émission de ce signal depuis une autre tâche. Il est possible de définir jusqu'à 32 signaux

différents par tâche.

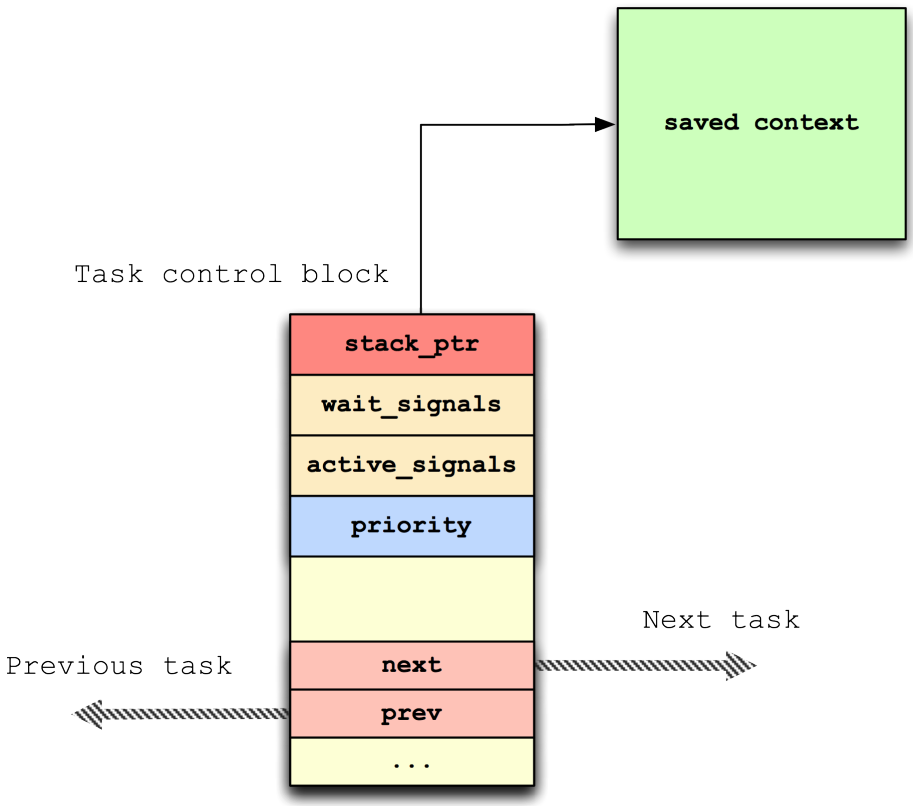

Figure 4. Structure de contrôle d'une tâche

Ce système par signaux est massivement utilisé pour la synchronisation dans l'échange des données entre les différentes tâches.

Suivant la nature des données, des structures différentes sont utilisées. Les données traitées par blocs sont allouées sur le tas et poussées dans des files FIFO partagées. La tâche DS [6] offrent aussi une API dédiée pour la création de *pipes* entre les tâches, pour la gestion des données par flot (telles que des flux réseau).

6. Data Services

La gestion des accès concurrents sur des ressources partagées est globalement de piètre qualité. Bien que le noyau supporte nativement les sections critiques, celles-ci ne sont pratiquement jamais utilisées au sein du système. Dans la très grande majorité des cas, le système optera pour le masquage total des interruptions ou l'emploi d'un verrou global sur l'ordonnanceur (prévenant ainsi tout changement de contexte).

Appels asynchrones REX implémente les appels de procédures asynchrones. Un appel asynchrone permet de différer l'exécution d'une routine dans le contexte d'une autre tâche en attente. La routine sera alors exécutée lorsque la tâche cible se réveillera.

La terminologie utilisée par REX est empruntée à celle du monde Windows, bien que l'implémentation sous-jacente soit assez différente. REX définit ainsi deux mécanismes :

- Les appels asynchrones (APC)
- Les appels différés (DPC)

Dans les deux cas, ces mécanismes permettent de différer l'exécution d'une routine. Le principe d'un APC consiste à pousser un nouveau contexte d'exécution sur la pile d'une tâche endormie, puis à provoquer un changement de contexte si besoin (figure 5).

Cependant l'appel à l'ordonnanceur n'étant pas possible depuis une routine d'interruption, REX définit une tâche DPC dont le seul rôle est de recevoir et distribuer les APC. Cette tâche s'exécute avec une priorité maximum sur le système pour amortir le changement de contexte nécessaire vers la tâche DPC (figure 6).

En résumé,

- les APC ne peuvent être utilisés que dans un contexte d'exécution normal.
- les DPC peuvent être utilisés dans le contexte d'une interruption mais engendrent une indirection en passant par la tâche DPC.

Timers REX supporte la création de *timers* pour l'exécution de routines à des intervalles de temps réguliers. Trois types d'actions peuvent être définis lors du déclenchement d'un timer :

- L'envoi d'un signal spécifique vers une tâche arbitraire
- L'exécution d'une APC dans le contexte d'une tâche arbitraire

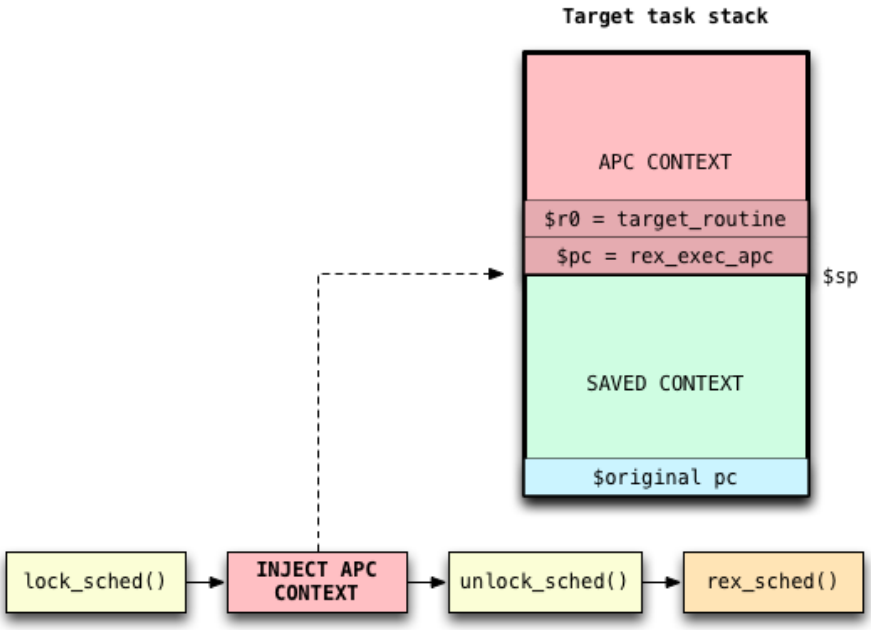

Figure 5. REX APC

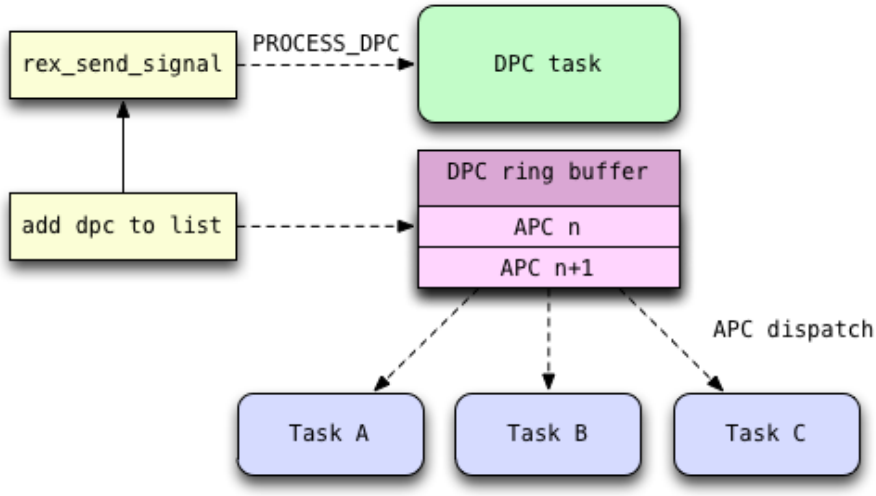

Figure 6. REX DPC

– L'exécution directe d'une routine dans le contexte courant (tâche MAIN ou routine d'interruption)

Allocation de mémoire La structure du tas est relativement simple. Les métadonnées d'un bloc sont constituées d'un pointeur vers le prochain bloc et d'un drapeau indiquant si le bloc est libre ou non. La structure principale du tas garde une trace du premier et dernier bloc alloué. L'allocation d'un bloc sur le tas est ainsi très peu coûteuse en temps. En revanche, le tas semble particulièrement sensible à la fragmentation. Le système se contente de fusionner régulièrement les blocs libres adjacents, mais alloue toujours les données après le dernier bloc alloué.

Bien que le système dispose d'un tas global, celui-ci est très peu utilisé. Chaque tâche initialise son ou ses propres tas lors de sa création.

4 Exécution de code arbitraire et débogage du système

4.1 Exécution de code

Une analyse statique de l'ensemble du système est possible, mais extrêmement fastidieuse. Le code est d'une très grande complexité et le système est par nature fortement concurrent.

Le protocole de diagnostic offre la possibilité d'écrire des données arbitraires n'importe où en mémoire. La MMU ne protège en lecture seule que les 13 premiers mégaoctets de la mémoire. Or le segment de code principal s'étant jusqu'aux 16 premiers mégaoctets de la mémoire. L'exécution de code est donc triviale à obtenir.

Pour communiquer avec le débogueur, il convient de s'injecter dans une tâche et réutiliser un canal de communication existant sur le port USB. Deux choix sont possibles :

– Le canal de communication AT (géré par la tâche DS)
– Le canal de la tâche de diagnostic (géré par la tâche DIAG)

Le choix de la tâche cible est critique puisqu'il ne sera ensuite pas possible de la déboguer. Mon choix s'est porté sur la tâche de diagnostic, celle-ci n'étant pas vitale au bon fonctionnement du système. Il est ainsi toujours possible d'analyser les gestionnaires de commandes AT.

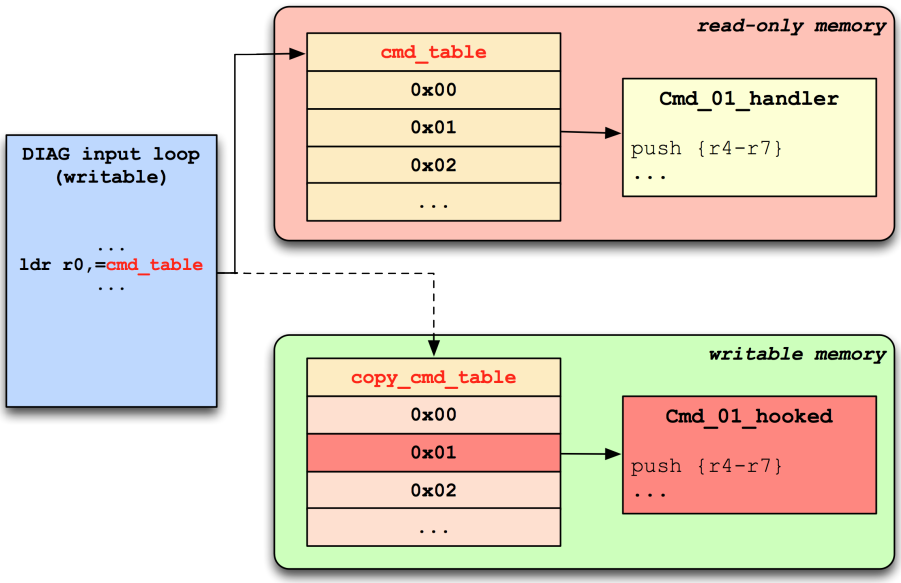

Figure 7. Détournement du flot d'exécution de la tâche DIAG

4.2 Arrêt et reprise d'une tâche

Le code s'exécute en mode ARM superviseur, nous disposons donc d'un contrôle complet sur la MMU et sur les vecteurs d'interruptions (nécessaires pour l'implémentation des points d'arrêt).

Le débogueur doit être en mesure de stopper et reprendre l'exécution d'une tâche. Pour parvenir à ce résultat, une interaction avec l'ordonnanceur du noyau est indispensable. Lorsque qu'une tâche s'arrête momentanément (suite à un point d'arrêt ou sur un ordre direct du débogueur), elle se met en l'écoute d'un signal partagé entre elle et le débogueur uniquement.

Lorsqu'un point d'arrêt est atteint :

1. Le gestionnaire de l'interruption prend la main
2. Un message est envoyé au client du débogueur sur l'interface USB
3. La tâche se met en attente du signal du débogueur

Le débogueur peut forcer une tâche à s'arrêter momentanément de la façon suivante :

1. Un appel asynchrone est exécuté dans le contexte de la tâche cible

2. La routine APC exécuté confirme l'arrêt de la tâche par un message sur l'interface USB
3. La tâche se met en attente du signal du débogueur

Le débogueur peut alors ordonner la reprise de l'exécution en envoyant à la tâche le signal adéquat. L'état de la tâche peut être récupéré et modifié via le contexte sauvegardé sur la pile.

4.3 Mode pas-à-pas

Contrairement aux architectures Intel, les processeurs ARMv5 ne disposent pas de fonctionnalités pour exécuter du code en mode pas-à-pas. Plusieurs implémentations logicielles sont possibles :

1. Prédire la prochaine instruction à exécuter, y apposer un point d'arrêt et continuer l'exécution. Cette solution, bien qu'étant la plus simple, peut poser des problèmes de concurrence sur un système multi-threads.
2. Emuler l'instruction dans le débogueur et modifier le contexte du thread courant.
3. Reloger l'instruction dans une zone mémoire séparée, et rediriger le pointeur d'instruction du thread courant sur cette zone.
4. Implémenter des addressages virtuels séparés pour chaque thread. Cette solution est la plus complexe à mettre en oeuvre puisqu'elle nécessite de modifier le code de l'ordonnanceur.

Bien que non-optimale, la première solution est actuellement celle utilisée par le débogueur.

4.4 Interface client

Le débogueur est finalement interfacé avec le protocole GDB[2]. Le débogueur récupère la liste des tâches du système et les présente à GDB sous forme de threads, en indiquant leur nom et l'état de leurs signaux.

```
(gdb) info threads
  Id   Target Id         Frame
  69   Thread 69 (REX DPC Task  [wait: 0x00000001; active: 0x00000000]) (running)
  68   Thread 68 (DOG           [wait: 0x00006800; active: 0x00000000]) (running)
  67   Thread 67 (DISP          [wait: 0x0000601b; active: 0x00000000]) (running)
  66   Thread 66 (SND           [wait: 0x0000636f; active: 0x00000000]) (running)
  65   Thread 65 (MDSP          [wait: 0x0000603f; active: 0x00000000]) (running)
  64   Thread 64 (FC            [wait: 0x0000001f; active: 0x00000000]) (running)
  63   Thread 63 (WCDMA L1      [wait: 0x03d97607; active: 0x00000000]) (running)
  62   Thread 62 (GSM L1        [wait: 0x0000080f; active: 0x00000400]) (running)
  61   Thread 61 (MGPCC         [wait: 0xffffffff; active: 0x00000000]) (running)
```

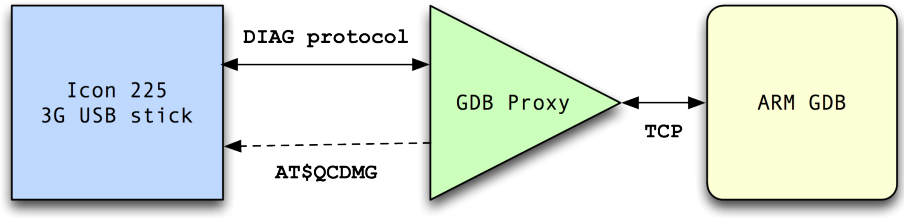

Figure 8. Architecture du débogueur

```
  60    Thread 60 (WCDMA L2 UL    [wait: 0x00017bff; active: 0x00000000]) (running)

  [snip snip]

  10    Thread 10 (SECDRM         [wait: 0x00006003; active: 0x00000000]) (running)
  9     Thread 9 (SECSSL          [wait: 0x000063cb; active: 0x00000000]) (running)
  8     Thread 8 (SEC             [wait: 0x0000778b; active: 0x00000000]) (running)
  7     Thread 7 (UIDC TSK        [wait: 0x000060db; active: 0x40000004]) (running)
  6     Thread 6 (GRAPH           [wait: 0x00006003; active: 0x00000000]) (running)
  5     Thread 5 (cb              [wait: 0x00000007; active: 0x00000000]) (running)
  4     Thread 4 (SECCRYPTARM     [wait: 0x00006003; active: 0x00000000]) (running)
  3     Thread 3 (GPS_FS          [wait: 0xffffffff; active: 0x00000000]) (running)
  2     Thread 2 (SLEEP           [wait: 0x00000000; active: 0x8f030002]) (running)
* 1     Thread 1 (REX Idle Tas    [wait: 0x00000000; active: 0x00000000]) (running)

(gdb) thread 68
[Switching to thread 68 (Thread 68)](running)

(gdb) interrupt
[Thread 68] #68 stopped.

(gdb) i r
r0             0xdc924c 14455372
r1             0xccc  3276
r2             0x154af3 1395443
r3             0xf059d0 15751632
r4             0x6800 26624
r5             0x0  0
r6             0x0  0
r7             0x168dc2c  23649324
r8             0xabab 43947
r9             0xabab 43947
r10            0x1889964  25729380
r11            0xabab 43947
r12            0x600000d3 1610612947
sp             0x188a120  0x188a120
lr             0x168dc5c  23649372
pc             0x137ab7 0x137ab7
cpsr           0x600000f3 1610612979

(gdb) x/5i $pc
   0x137ab7:  ldr r0, [r7, #48] ; 0x30
   0x137ab9:  cmp r6, #0
   0x137abb:  ldr r4, [r0, #12]
   0x137abd:  str r5, [r7, #32]
   0x137abf:  bne.n 0x137ac4
```

Listing 1.1. Exemple de session GDB

5 Instrumentation avancée

5.1 Implémentation des tracepoints

L'utilisation seule de points d'arrêt est rapidement limitée face à ce type de système. Plusieurs problèmes peuvent se poser :

- Un point d'arrêt dans une routine utilisée par le débogueur ou dans un contexte d'interruption tuera la session de déboguage
- Stopper l'exécution d'une tâche peut engendrer le gel d'autres tâches

Bien souvent, l'arrêt d'une tâche n'est pas nécessaire. Il est préférable de détourner son flot d'exécution, enregistrer les informations dont nous avons besoin (registres, pile...) et aussitôt lui laisser reprendre son exécution normalement.

Pour exécuter des actions complexes sans redonner la main au débogueur, GDB compile ses expressions dans un *bytecode*[1] qui sera exécuté par le *stub*. Le débogueur embarque donc une petite machine virtuelle capable d'interpréter le bytecode de GDB.

Lorsqu'un tracepoint est atteint :

1. La main est passée au gestionnaire de l'interruption
2. L'action du tracepoint est interprétée dans la machine virtuelle. Cette action peut enregistrer dans un tampon diverses informations comme l'état des registres, des parties de la mémoire, etc.
3. Le point d'arrêt ne peut être supprimé, nous redonnons donc la main vers un tampon où l'instruction écrasée a été précédemment relogée.
4. L'exécution continue ensuite à l'instruction suivante [7].

GDB ne supporte pas le relogement des instructions Thumb. Malheureusement, la quasi-intégralité du code est compilé en mode Thumb. Le débogueur embarque donc aussi un relogeur d'instruction Thumb. Le relogement d'une instruction est plus complexe que la simple copie de l'instruction en mémoire. L'instruction doit être modifiée si celle-ci adresse ou modifie le registre PC.

A titre d'exemple, l'instruction :

```
LDR R0, [PC, #4]
```

Listing 1.2. Instruction Thumb

7. qui n'est pas nécessairement l'instruction située après l'instruction originale

est transformée en

```
PUSH {R1}
LDR R1, [ORIGINAL PC]
LDR R0, [R1, #4]
POP {R1}
LDR PC, [ORIGINAL PC]
```

Listing 1.3. Instruction Thumb relogée

Le fonctionnement général des tracepoints est décrit sur la figure 9.

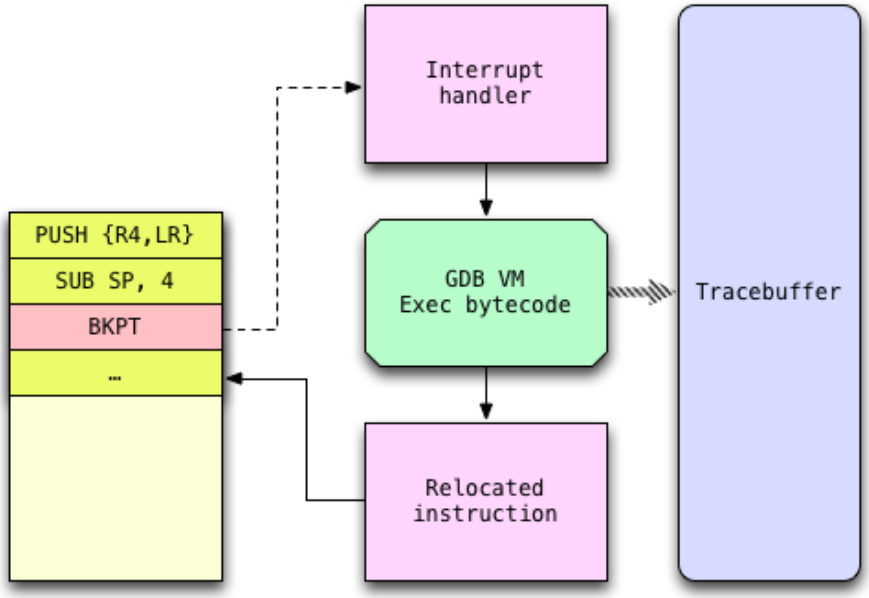

Figure 9. Fonctionnement des tracepoints

Exemple Dans cet exemple, nous posons un tracepoint sur la fonction de REX chargée d'envoyer un signal vers une tâche. Le prototype de cette fonction est le suivant : `int rex_send_signal(rex_task * task, int signals);`. Le premier argument est récupérable dans le registre `R0` et le signal en question dans le registre `R1`. Nous récupérons aussi le nom de la tâche appelante, en déréférençant l'adresse `0x168dc5c` qui contient le pointeur vers la structure de la tâche courante.

```
(gdb) trace *0x137572
(gdb) passcount 3
(gdb) actions
>collect (*(rex_task **)0x168dc5c)->name
>collect ((rex_task *)$r0)->name
>collect $r1
>end

(gdb) tstart
(gdb) tstop

(gdb) tfind
Found trace frame 0, tracepoint 1
#0  0x00137572 in ?? ()
(gdb) tdump
Data collected at tracepoint 1, trace frame 0:
(*(rex_task **)0x168dc5c)->name = "WCDMA L1\000\000\000"
((rex_task *) $r0)->name = "WCDMA L1\000\000\000"
$r1 = 4

(gdb) tfind
Found trace frame 1, tracepoint 1
#0  0x00137572 in ?? ()
(gdb) tdump
Data collected at tracepoint 1, trace frame 1:
(*(rex_task **)0x168dc5c)->name = "WCDMA L1\000\000\000"
((rex_task *) $r0)->name = "SLEEP\000\000\000\000\000\000"
$r1 = 2
```

Listing 1.4. Exemple d'utilisation des tracepoints

Dans cet exemple, nous pouvons voir que la tâche gérant la couche 1 du protocole WCDMA (3G) envoie un signal à la tâche SLEEP (tâche d'inaction). Etant donné que cette capture a été réalisée sur une clé déconnectée du réseau, nous pourrions supposer que le signal 2 correspond à une demande de passage en mode économie d'énergie.

5.2 Implémentation des shellpoints

L'implémentation des tracepoints de GDB souffre de plusieurs limitations. Le bytecode GDB n'offre par exemple pas la possibilité de modifier la mémoire, ni d'appeler une fonction arbitraire. Il pourrait pourtant s'avérer utile de pouvoir appeler une fonction pour savoir, par exemple, l'identité de la tâche qui a touché le tracepoint. Pour cette raison, j'ai introduit la notion de *shellpoints* dans le débogueur. Un shellpoint fonctionne comme un tracepoint, mais exécutera un *shellcode* natif en lieu et place du bytecode de GDB.

L'implémentation finale des shellpoints reste à définir. Il est probable que

le choix de l'assembleur se porte sur Miasm[3], intégré directement dans GDB Python.

6 Pistes d'investigation futures

6.1 Extraction des flux réseau

Le débogueur ayant accès à toute la mémoire du système, il devrait être possible d'extraire les flux réseau transitant sur la clé. Cependant, le suivi des données au sein du baseband est plutôt complexe, celles-ci transitant rapidement d'une tâche à l'autre. Un axe d'analyse possible serait de se concentrer sur des fonctions de manipulation de données génériques du système. En utilisant le système de tracepoints, l'idée serait de mettre en observation les données circulant dans les *pipes* entre les tâches, d'extraire leur contenu, puis de séparer le grain de l'ivraie pour retrouver une partie des trames réseau.

6.2 Fuzzing local des piles protocolaires

Le fuzzing des piles protocolaires téléphoniques est un sujet complexe. Les travaux existants sur le sujet[18,11,12,8,17] sont clairement orientés sur le protocole GSM et utilisent des stations de base modifiées pour forger des trames corrompues.

Deux avantages se présentent avec le débogueur :

- Il est possible de forger des données en local, et de les passer directement au système en appelant les bonnes fonctions, sans utiliser de station de base.
- Il est théoriquement possible de fuzzer des piles protocolaires 3G. OpenBTS ne supportant pas la 3G, les implémentations de ces protocoles ont fait l'objet de peu d'attaques jusqu'à présent[6].

6.3 Analyse des DSP

La clé 3G ciblée dans cet article fait usage de deux DSP propriétaires à Qualcomm. Les spécifications et le jeu d'instructions de ces DSP sont non-documentés. Obtenir le contrôle de ces DSP peut présenter plusieurs intérêts :

- Analyser l'implémentation de la cryptographie
- Récupérer l'algorithme de chiffrement du GPRS
- Récupérer le signal à destination d'autres mobiles
- etc.

6.4 OKL4

Les basebands Qualcomm récents n'utilisent plus REX comme noyau temps-réel principal. REX est à présent « virtualisé » dans OKL4, un dérivé propriétaire de L4::Pistachio. Analyser le fonctionnement de ce noyau s'avère indispensable pour un potentiel portage du débogueur sur ces plateformes. Les téléphones récents font déjà l'objet de beaucoup d'attention dans les cercles de passionnés[13,7,16] et dans la communauté du SIM-unlocking[10,9].

7 Conclusion

Bien que les basebands Qualcomm plus récents présentent une architecture légèrement différente, la clé 3G Option Icon 225 représente un bac à sable intéressant :

- obtenir l'exécution de code est relativement aisé, avec un contrôle total du CPU et de la mémoire
- le microcode de la clé n'est pas signé et peut donc potentiellement être remplacé
- une grande partie du code est commune avec les modèles plus récents

L'objectif a moyen terme du projet est de pouvoir contrôler les flux réseau transitant dans la clé, en particulier pour la 3G. Le débogueur constitue aussi un outil intéressant en vue de rechercher des vulnérabilités, celui-ci étant à même d'intercepter les exceptions déclenchées par des morceaux de code vulnérables.

Le code du débogueur est disponible sous licence GPL sur Google Code[5].

Références

1. Gdb documentation : Bytecode descriptions. http://sourceware.org/gdb/onlinedocs/gdb/Bytecode-Descriptions.html.
2. Gdb remote serial protocol. http://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html.
3. Miasm, reverse engineering framework. http://code.google.com/p/miasm.
4. Osmocombb project. http://bb.osmocom.org.
5. qcombbdbg, a qualcomm baseband debugger. http://code.google.com/p/qcombbdbg.
6. Ravishankar Borgaonkar, Nico Golde, and Kevin Redon. Femtocells : A poisonous needle in the operator's hay stack. *Black Hat USA*, 2011.

7. XDA developers. Communauté de développement et hacking sur mobiles. http://forum.xda-developers.com.
8. Brinio Hond. Fuzzing the gsm protocol. Master's thesis, Radboud University Nijmegen, 2011.
9. Musclenerd (iPhone Dev Team). Evolution of the iphone baseband and unlocks. *Hack In The Box security conference, Amsterdam*, 2012.
10. Luis Miras. The baseband playground. *EkoParty conference*, 2011.
11. Colin Mulliner and Nico Golde. Sms-o-death. 27^{th} *Chaos Computer Congress*, 2010.
12. Collin Mulliner and Charlie Miller. Fuzzing the phone in your phone. *Black Hat USA*, 2009.
13. Tim Newsham. Rétroconception de iguana/l4 sur htc dream. http://code.google.com/p/docl4amss/wiki/Main.
14. Karsten Nohl and Luca Melette. Defending mobile phones. 28^{th} *Chaos Computer Congress*, 2011.
15. Karsten Nohl and Sylvain Munaut. Wideband gsm sniffing. 27^{th} *Chaos Computer Congress*, 2010.
16. TJWorld. Wiki d'information sur la rétroconception sur mobiles. http://tjworld.net.
17. Ralf-Phillip Weinmann. The baseband apocalypse. 27^{th} *Chaos Computer Congress*, 2010.
18. Harald Welte. Gsm protocol fuzzing and other gsm related fun. *0sec conference*, 2009.
19. Harald Welte. Running your own gsm stack on your phone. 27^{th} *Chaos Computer Congress*, 2010.
20. Harald Welte. Cellular protocol stacks for internet. 28^{th} *Chaos Computer Congress*, 2011.