Formation à Linux Embarqué

Jérôme Pouiller <j.pouiller@sysmic.org>

Quatrième partie IV

Debuguer

Debuguer une application utilisateur

Utilisation classique. `-g3` permet d'inclure les symboles de debug

```
1 host$ gcc -Wall -g3 -c hello.c -o hello.o
  host$ gcc -Wall -g3 hello.o -o hello-x86-dbg
  host$ gdb hello-x86-dbg
```

Debuguer une application utilisateur

Commandes utiles

Gestion de l'éxecution

- Point d'arrêt à l'adresse `0x9876`

```
gdb> b *0x9876
```

- `b hello.c:30` Point d'arrêt à la ligne 30 du fichier `hello.c`

```
gdb> b hello.c:30
```

- Point d'arrêt sur la fonction `main`

```
gdb> b main
```

- Continuer un programme arrêté

```
gdb> c
```

- Démarrer le programme avec `arg` comme argument

```
gdb> r arg
```

Debuguer une application utilisateur

Commandes utiles

Obtenir des informations

- Voir la pile d'appel

```
gdb> bt
```

- Afficher les variable locales à la fonction

```
gdb> i locals
```

- Afficher les arguments de la fonction

```
gdb> i args
```

- Afficher la valeur de `a`

```
gdb> p a
```

- Afficher la valeur de `abs(2 * (a - 4))`

```
gdb> p abs(2 * (a - 4))
```

Debuguer une application utilisateur

Commandes utiles

Debuguer à chaud

- Attacher `gdb` à un processus existant

```
gdb> attach 1234
```

- Arrêter le debug sans arrêter le processus

```
gdb> detach
```

Debuguer une application utilisateur

Commandes utiles

Et plus...

- Afficher la valeur de `i` à chaque fois que l'on passe sur le point d'arrêt 1

```
gdb> command 1
> silent
> p i
> c
> end
```

- Arrêter le programme lorsque la variable `i` est modifiée ou lue

```
gdb> watch i
gdb> awatch i
```

- Passer d'une thread à une autre

```
gdb> thread 2
gdb> thread 1
```

Debuguer une application utilisateur

Utilisation des Cores Dump :

- Vérifiez la limite *coredump size* avec `ulimit -c`
- Core Dump automatique en cas de crash
- Forcé avec `kill -QUIT` ou `<Ctrl+\>` (man kill)
- Créé avec la commande gdb `gcore` (voir avec attach) (malheureusement, ne fonctionnement pas en mode debug croisé)
- Il est possible de savoir à quelle binaire est attaché le coredump avec `file`

Debuguer une application utilisateur distante

Utilisation croisée

- Compilation

```
host$ arm-linux-gcc -g3 -c hello.c -o hello.o
host$ arm-linux-gcc -g3 hello.o -o hello-arm-
    dbg
```

- Démarrage du serveur de debug

```
target$ gdbserver *:1234 hello
host$ arm-ulibceabi-gdb hello
```

- Configuration des chemins

```
> set solib-absolute-prefix /home/user/nfs
```

- Connexion au serveur de debug

```
> target remote target:1234
```

Debuguer une application utilisateur distante

Debug d'une application en cours d'éxecution

- Execution

```
target$ ./hello > /dev/null &
```

- On attache `gdb` au processus

```
target$ gdbserver *:1234 --attach $!
```

- configuration classque de `gdb`

```
host$ arm-linux-gdb
> exec-file hello
> set solib-absolute-prefix /home/user/nfs
> target remote target:1234
```

- Remarque : Si vous dugger une programme utilisant `libpthread`, vous devez avoir le fichier `libthread_db` pour que `gdb` puisse décoder les emplacements les piles des threads
- Libérons le processus

```
> detach
```

Debuguer une application utilisateur

Séparer les symbols de debug permet :

- de n'uploader sur la cible que le nécéssaire à l'éxecution.
- de livrer des binaires strippées en production tout en conservant les possibilités de debug
- d'analyser des *core dump* produit en production

Utilisation de `objcopy` pour créer le fichier de symboles :

```
host$ arm-linux-objcopy --only-keep-debug hello
   hello.sym
host$ chmod -x hello.sym
host$ arm-linux-strip --strip-unneeded hello
target$ gdbserver *:1234 hello
host$ arm-linux-gdb
> target remote target:1234
> symbol-file hello.sym
```

Debuguer une application utilisateur

Sous certaines conditions, il est possible de rendre le chargement des symboles automatique :

```
host$ arm-linux-objcopy --add-gnu-debuglink=hello.
    sym hello
```

Remarque : l'option -g n'influence pas le code compilé :

```
$ gcc hello.c -O3 -o hello
$ gcc hello.c -O3 -g -o hello.dbg
$ strip hello.dbg hello
$ cmp hello.dbg hello && echo ok
ok
```

Analyse sans execution

Très bons outils pédagogiques

- `strings` affiche les chaine de caractères affichables d'une binaire
- `ldd` affiche quel sera le comportement de `ld` lors du chargement (en d'autres termes, liste les dépendances avec des bibliothèques dynamique)
- `gdb` est capable de charger des fichiers objets et d'en sortir beaucoup d'informations sans execution
- `objdump -h` permet d'obtenir des information sur le placement des sections dans le fichier
- `objdump -T` ou `readelf -s` ou `nm` liste les symbols contenu dans un fichier
- `objdump -d` permet de désassembler un fichier objet
- `objdump -S` permet d'obtenir l'assembleur intermixé avec les code
- `objdump -C` permet de "demangler" des fonction C++
- `adddr2line` permet de convertir une addresse mémoire en numéro ligne dans le code

Debuguer le noyau

Activiation d `kgdb` permettant d'embarquer `gdbserver` dans le noyau :

- Compilation

```
host$ cp -a build build-dbg
host$ make O=build-dbg ARCH=arm CROSS_COMPILE=
    arm-linux- menuconfig
... Compile the kernel with debug info ...
... KGDB: kernel debugging with remote gdb ...
host$ make -j3 O=build-dbg ARCH=arm
    CROSS_COMPILE=arm-linux- uImage
```

- L'image ELF est beaucoup plus grosse

```
host$ ls -l build-dbg/vmlinux build/vmlinux
host$ cp build-dbg/uImage /srv/tftp/uImage
    -2.6.33.7-dbg
```

Debuguer le noyau

- Passage des arguments `kgdboc` indiquant quel interface `kgdb` doit utilisé et `kgdbwait` indiquant que kgdb doit attendre notre connexion avant de continuer le boot

```
uboot> set bootargs ${bootargs} kgdboc=ttyS0
    kgdbwait
uboot> tftp
<Ctrl-a q>
host$ arm-linux-gdb vmlinux
gdb> target remote /dev/ttyUSB0
```

Kgdb et les modules

- Les modules sont chargé dynamiquement. Ils ne sont pas contenu dans le fichier `vmlinux`
- Il est nécessaire d'indiquer à gdb de charger le module que l'on souhaite débuguer
- Néanmoins, gdb ne peut pas savoir à quel addresse à été chargé le module
- Ces informations peuvent être récupérées dans `/sys/modules/*/sections`
- On peut alors les donner à gdb avec `add-symbol-file` :

```
target% cat /sys/module/mod2_chr/sections
host$ gdb vmlinux
> target remote 192.168.1.55:2345
> add-symbol-file my_modules.ko 0xd0832000 \
    -s .bss 0xd0837100 -s .data 0xd0836be0
```

- Il existe des scripts effectuant la procédure automatiquement. A adapter suivant vos besoins.
- Remarque : Les modules sont des fichiers ELF normaux. Il

ld.so

Lors du chargement d'une binaire, la bibliothèque `ld.so` charge les bibliothèques indiquées dans la section *.dynamic*. Il est possible de changer le comportement de `ld.so` par des fichier de configuration (`/etc/ld.so.conf`) ou par des variables d'environnement.

- `LD_LIBRARY_PATH` permet d'ajouter un répertoire ou les bibliothèques seront recherchées.
- `LD_PRELOAD` permet de précharger une bibliothèque. Ainsi les symboles définis dans cette bibliothèque surcharge les symboles standards

ld.so

```
#include <stdio.h>
#include <unistd.h>
unsigned int sleep(unsigned int s) {
   static int (*real_sleep)(unsigned int) = NULL;
   if (!real_sleep)
      real_sleep = dlsym(RTLD_NEXT, "sleep");

   printf("Try to reveal race-conditions\n");
   return real_sleep(5 * s);
}
```

ld.so

```
host$ gcc -shared -ldl -fPIC mysleep.c -o
    libmysleep.so
target$ LD_PRELOAD=libmysleep.so ./hello
target$ export LD_PRELOAD=libmysleep.so
taregt$ ./hello
```

- Cette technique est utilisé pour tracer et profiler certains évènement
- Les expert en sécurité l'utilisent pour détourner un programme de son but original
- Enfin, il permet de simuler des problèmes afin de vérifier la robustesse d'un programme

strace et ltrace

- `strace` permet de d'afficher les appels systèmes effectué par un processus. Il a l'avantage d'afficher beaucoup d'informations sous des formes lisible.

```
target$ strace ./hello
```

- Il est possible de filtrer les appels systèmes à suivre.
- `ltrace` est une génératlisation de `strace`. Il permet de suivre les appels à une bibliothèque extérieure

```
host$ arm-linux-ldd --root . ./hello
target$ ltrace -l /lib/libc.so.0 -e sleep ./
    hello
target$ ltrace -l /lib/libc.so.0 -e malloc -e
    free /bin/ls
```

- Ces outils sont une aide préciseuse lorsque l'on souhaite identifier le comportement d'une binaire dont nous ne disposons pas des sources.

ptrace

`ptrace` est l'appel système permettant le fonctionnement de debuguers et d'outils tel que `strace` et `ltrace`.

Il permet de prendre la main sur un processus, puis d'obtenir des informations sur son état et de modifier sa mémoire.

Couverture de code

Principe :

- On instrumente le code généré
- Entre chaque ligne de C (équialent à plusieurs ligne d'assembleur), on ajoute une instruction du type `inc $ADDR`
- `ADDR` est une constante definie à la compilation
- `ADDR` pointe sur une entrée d'un tableau initialisé à zero
- Chaque entrée du tableau symbolise une ligne
- Par conséquent, 100000 ligne augmentra la taille en mémoire de l'application d'environ 400Ko
- Lorsque le programme se termine, on écrit le tableau sur le disque (fichiers `.gcda`)
- Un fichier index est généré durant la compilation afin de faire le lien entre les lignes du fichier source et les données générées (fichiers `.gcno`)

Fonctionne sur une binaire utilisateur. Permet de valider une campagne de test unitaire.

Couverture de code

Utilisation de `gcov` en natif :

- Compilation

```
host$ mkdir x86-cov
host$ gcc --coverage hello.c -o x86-cov/hello
```

- Execution d'un scénario d'utilisation

```
host$ x86-cov/hello
```

- Conversion des fichiers `gcda` dans un format lisible

```
host$ for i in **/*.gcda; do
> gcov -o $(dirname $i) $i
> done
```

- Convertion des fichier `gcda` en HTML en utilisant `lcov/genhtml`

```
host$ lcov -i -o coverage-x86.info -b . -d x86-
    cov
host$ genhtml -o html coverage-x86.info
```

Couverture de code

Utilisation de gcov sur la cible :

- On précise le répertoire ou les fichier doivent être stockés

```
host$ cd arm-cov
host$ arm-linux-gcc --fprofile-generate=/tmp/
    data --coverage ../hello.c -o hello-arm-cov
```

- On execute sur la cible

```
host$ scp hello-arm-cov root@target:
target$ ./hello-arm-cov
```

- On récupère les résultats pour les traiter sur le host

```
host$ scp -r root@target:/tmp/data .
```

Couverture de code

- Conversion des fichiers `gcda` dans un format lisible

```
host$ for i in **/*.gcda; do
> arm-linux-gcov -o $(dirname $i) $i
> done
host$ lcov -i -o coverage-arm.info -b .. -d
    data --gcov-tool arm-linux-gcov
host$ genhtml -o html coverage-arm.info
host$ firefox html/index.html
```

Fonctionne aussi avec le noyau : `CONFIG_LCOV`

Valgrind

On execute notre programme dans une machine virtuelle. Cette machine virtuelle

- instrumente les accès en lecture et en écriture à la mémoire.
- suit les appels aux fonctions `malloc` et `free`
- suit les copie de pointeurs

Ainsi, en plus de détecter les zone non désallouées à la sortie du programme, `valgrind` est capable de détecter :

- l'écriture et la lecture d'une zone non allouée (voir aussi *DUMA*)
- la lecture d'une zone jamais écrite
- les zone de mémoire n'étant plus pointées

`valgrind` peut indiquer les contextes exacts où tous ces évènements se produisent.

Valgrind

```c
/*
 * Test with:
 *  echo 123456 | valgrind --leak-check=yes ./a.out
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void fun() {
  int      var;
  int      *stck = &var;
  int      *heap = malloc(2 * sizeof(int));
  static int *bss;

  printf("%d\n", var);     // var hasn't been
     initialized
```

Valgrind

Appellons nos deux fonctions de tests

```
heap[2] = 42;                    // wrote past the end of
    the block
printf("%d\n", heap[-1]); // read from before
    start of the block
printf("%d\n", stck[-15]); // reading from a bad
    stack location (detected if outside current
    frame)
read(0, heap, 10);               // buffer passed to
    syscall is too small
//read(0, stck, 10);             // buffer passed to
    syscall is too small (not detected)
free(heap);
printf("%d\n", *heap);   // heap was already freed
heap = malloc(sizeof(int)); // memory leaks (
    definitely lost)
stck = heap;                     // .. even if another
    variable point on it
```

Valgrind

Valgrind a donné naissance à d'autres outils :

- Cachegrind permet de simuler les lignes de cache du processeur et ainsi d'indiquer ou se produise les erreurs de pages (voir aussi `perf`)
- Callgrind permet d'instrumenter les appels de fonctions. Il génère des graphes d'appels et des informations de profiling (voir aussi `perf`)
- Hellgrind et DRD instrumente les mécanismes générant des sections critique et détecte les *Race Conditions* sur les données.
- Massif detecte les contexte d'allocation de mémoire (heap profiling)
- Ptrcheck suit les blocks pointé et détecte ainsi les overruns (voir aussi *DUMA*)

Bases

Avant de se lancer dans des procédure de profiling complexe, quelques commandes simples :

- `free` indique les ressources mémoire disponibles sur le système
  - Les *buffers* indiquent des données devant être écritent sur le disque (ou sur un IO) mais dont le système repouse l'écriture pour des raisons de performances
  - Les *caches* indiquent des données lue sur le disque (ou sur un IO) conservées en mémoires afin d'accélérer les futurs accès
- `time` indique les ressources utilisées par un processus
  - Temps total
  - Temps CPU en utilisateur et dans le noyau
  - Nombre de d'erreur de page majeure (nécessitant un accès disque) et mineurs
  - Nombre de message socket (réseau) envoyés/reçus
  - Mémoire utilisée en moyenne
  - Nombre d'accès aux disques en entrée et en sortie

Bases

- `top` indique à peu près les même informations que time mais de manière intéractive
- `ps` affiche aussi ces informations sur un ou plusieurs processus en cours d'éxecution
- `/proc/${PID}/{stat,statm,status}` contiennent les données bruts de ces outils
- `/proc/${PID}/{map,smap,pagesmap}` contiennent des infomration très précisesnt aux sujet du mapping mémoire d'un processus

perf

`perf` utilise les registre de debug du cpu. Faible overhead. depend du CPU. Le cpu le plus fourni est x86.

- Obtenir une vue synthétique d'un processus

```
target% apt-get install linux-tools-common
target$ perf stat ./hello-dbg
```

- Effectuer une mesure plus précise

```
target$ perf record ./hello-dbg
target$ perf report
```

- Annoter le code avec les information de profiling

```
target$ perf annotate timeattack
```

perf

- Même manipulation mais avec les contextes d'appel

```
target$ perf record -g ./hello-dbg
target$ perf report
```

- Il est possible d'utiliser d'autre point de mesure que la consommation CPU

```
target$ perf list
target$ perf stat -e cache-misses ./hello-dbg
target$ perf record -e cache-misses ./hello-dbg
target$ perf report
target$ perf annotate timeattack
target$ perf record -e cache-misses -g ./hello-
    dbg
target$ perf report
```

oprofile

- Idem `perf`, mais moins générique
- Initialement sur les architectures ST (pas ARM, ni x86)
- Si vous avez un noyau suffisament récent, `oprofile` utilise `perf` comme backend
- Démarrage du daemon

```
target% opcontrol --init
```

- Liste des évènements pouvant être enregistrés

```
target% opcontrol -l
```

- Profiling d'un évènement

```
target% opcontrol --start --no-vmlinux -e
   CPU_CLK_UNHALTED:100000:0:1:1
target% ./hello
target% opcontrol --shutdown
```

oprofile

- Génération du rapport

```
target% opreport hello
```

- Génération du source annoté

```
target% opannotate --source hello-prof
```

U-boot

Notre vendeur nous fourni une version de u-boot patchée pour notre carte. Dans ce cas, il nous suffit de configurer correctement u-boot et de le compiler :

```
make usb-a9260_config
make CROSS_COMPILE=arm-linux-
```

U-boot

Si nous voulons ajouter une nouvelle carte, nous devons :

- Ajouter un nouveau fichier d'entêtes dans `include/configs`
- Ajouter un nouveau répertoire dans `board`
- A priori, vous n'aurez pas besoin d'ajouter une architecture et encore moins un jeu d'instruction
- Ajouter une cible dans le Makefile
- Cette cible appellera `mkconfig`
- `mkconfig` prend en paramètre le nom du fichier d'entête, le type de CPU, d'architecture et de carte, ainsi que le nom du vendeur et le modèle de chipset.

```
./mkconfig usb-a9260 arm arm926ejs usb-a9260
   Calao at91sam926x
```

U-boot

- `mkconfig` créera un fichier `config.h` pointant sur votre fichier entête et un fichier `config.mk` contenant :

```
ARCH   = arm
CPU    = arm926ejs
BOARD  = usb-a9260
VENDOR = Calao
SOC    = at91sam926x
```

- Placer les drivers nécéssaire à votre carte dans `board/usb-a9260`. L'implémentation des drivers est proche de ce que l'on trouve dans le noyau Linux. Il suffit d'associer les callback aux champs d'une structure.
- Le fichier `usb-a9260.h` devra contenir la configuration des fonctionnalité de u-boot
- Vous trouverez des options de configuration haut niveau telles que les commandes devant être incluse
- Vous trouverez aussi les paramètre d'initialisation bas niveau pour le CPU et l'architecture (timing mémoire, etc...)

OpenOCD

Qu'est-ce qu'une sonde Jtag ?

- TAP (Test Access Port) est un module matériel capable de traiter des commande de debug (souvent sur le processeur). Notre TAP est embarqué sur le ARM9260EJ-S.
- JTAG est un standard de communication normalisé pour les TAP.
- Une "hardware interface dongle" doit être utilisée pour empaqueter le standard JTAG dans un autre protocole (ethernet, USB, serie...) afin d'être traitable par un PC. Nous utilisons la FTDI FT2232 pour cette partie. Cette puce permet de présenter à l'host deux périphérique USB : Le JTAG et la console RS232.

OpenOCD

Qu'est-ce qu'une sonde Jtag ?

- OpenOCD permet de gérer la communication avec la puce FTDI et de traduire le langage de bas niveau JTAG en langage de debug gdb
- On trouve souvent des sonde JTAG incluant déjà l'équivalent de OpenOCD et sachant déjà communiquer avec gdb (par ethernet ou série très souvent). (Abatron, Lautherbar, etc...)
- On peut reconnaitre les deux types de sondes à leur prix. Les première coute au maximum quelques centaines d'euros alros que les secondes coutent rarement moins de 1000euros.

OpenOCD

Beaucoup de sondes JTAG (USB, port parallèle, port serie) sont supportées par OpenOCD : `doc/openocd/JTAG-Hardware-Dongles.html`.

```
host% apt-get install openocd
```

OpenOCD

Il est nécessaire de corriger un bug dans le fichier de configuration de notre cible :

```
host% ln -s at91sam9260.cfg /usr/share/openocd/
    scripts/target/at91sam9260minimal.cfg
```

Nous pouvons maintenant utiliser notre sonde Jtag

```
host$ openocd -f interface/calao-usb-a9260-c01.cfg
host$ telnet 127.0.0.1 4444
> help
> reset init
> arm reg
> arm disassemble 0x00000000 5
> arm disassemble 0x00000058 5
> ^C
```

OpenOCD avec gdb

- Lancement de `gdb`

```
host$ arm-linux-gdb
```

- Connexion avec OpenOCD

```
gdb> target remote localhost:3333
```

- `monitor` permet d'envoyer des commande brutes à OpenOCD

```
gdb> monitor reset init
gdb> monitor help
```

- Vu que nous ne sommes pas lié à une binaire, nos commandes sont limitées. Nous pouvons néanmoins dumper la mémoire et placer des breakpoints

```
gdb> x/10i 0
gdb> hbreak *0x58
gdb> b *0x0
```

OpenOCD

Deux mots sur la configuration :

- Les fichiers de `interfaces/` contiennent les informations sur le driver à utiliser et comment s'y connecter. Le repertoire `target/` contient la configuration du microcontrolleur en lui-même.
- La documentation se trouve sur `http://openocd.sourceforge.net/doc/openocd.html` ou dans `/usr/share/doc/openocd`
- Il est possible d'avoir l'aide contextuelle sur une commande `help command`

Debuguer U-Boot

- Notre fichier de configuration de OpenOCD ne lance pas la cible à la vitesse maximum. Laissons cette tâche à u-boot, puis arrêtons la cible.

```
Hit any key to stop autoboot: 0
u-boot>
```

- Arrêtons la cible afin de pouvoir travailler dessus

```
telnet> halt
```

- Configurons une vitesse de transmission raisonnable

```
telnet> jtag_hkz 10000
telnet> arm7_9 dcc_downloads enable
```

- Chargeons l'image ELF de u-boot

```
telnet> load_image .../u-boot (elf,13secondes)
```

Debuguer U-Boot

- Vérifions que nous retrouvons bien le code assembleur de U-boot à l'endroit indiqué

```
telnet> disassemble XXXX
```

- Sautons jusqu'à cette addresse

```
telnet> resume XXX
```

- Arrêtons le démarrage

```
telnet> halt
```

Debugguer U-Boot

Maintenant que notre cible est correctement initialisées, nous pouvons prendre la main dessus avec gdb :

```
host$ arm-linux-gdb u-boot
gdb> target remote localhost:3333
gdb> c
^C
gdb> hbreak run_command
gdb> c
u-boot> help
```