OPERATING SYSTEMS UE BONUS BEISPIEL

Vorbereitung

Zur Vorbereitung lesen Sie bitte die ersten drei Kapitel sowie Kapitel 5 des Buches „Linux Device Drivers“ [1]. Weiters steht Ihnen ein rudimentäres *„Hello World“* Modul inkl. Makefile in der Entwicklungsumgebung zur Verfügung. Dieses können Sie gerne als Grundlage für die Entwicklung Ihres eigenen Kernel-Moduls verwenden.

Abgabegespräch

Das Abgabegespräch findet am 24.01.2018 im TI-Labor statt. Eine Anmeldung in TUWEL zu einem Slot ist erforderlich - bitte bedenken Sie, dass eine nachträgliche Abgabe **nicht** möglich ist. Es gelten die bekannten Richtlinien mit den zusätzlichen Einschränkungen bzw. Lockerungen:

- Ihre Implementierung muss in der TI-Lab Umgebung (User Mode Linux) demonstriert werden.
- GNU C Standard (C99 Standard mit GNU Erweiterungen)
- Kernel Coding Style konform (siehe <kernel source dir>/Documentation/CodingStyle), Coding Style Check Tool (auch in der Entwicklungsumgebung verfügbar):

  ```
  # /usr/src/linux/scripts/checkpatch.pl -f <source file>
  ```

- Das Kernel-Modul muss sich sauber (d.h. Freigabe aller verwendeten Ressourcen) aus dem System per *rmmod(8)* entfernen lassen.
- Definieren Sie den Modul-Parameter `debug`, der, wenn er beim Laden des Moduls auf `1` gesetzt wird, für sinnvolle Debugausgaben (z.B. bei den einzelnen Operationen auf den Character Devices) sorgt.

Secvault, a Secure Vault

Implementieren Sie ein Linux Kernel Modul, welches über Character Devices bis zu 4 flüchtige, aber sichere Speicher (Secure Vaults - kurz: *Secvaults*) zur Verfügung stellt. Ein *Secvault* hat eine `id` und eine konfigurierbare Grösse (`size`) zwischen 1 Byte und 1 MByte. Er soll per eigenem Character Device vom Userspace aus beschreib- und lesbar sein. Daten, die auf ein *Secvault* Character Device geschrieben werden, sollen verschlüsselt als Cyphertext im Speicher abgelegt werden. Beim Lesen soll der im Speicher abgelegte Cyphertext entschlüsselt werden und als Klartext in den Userpace übergeben werden. Als Verschlüsselung soll ein einfaches symmetrisches *XOR*-Verfahren dienen: Abhängig von der Position im *Secvault*, wird ein Schlüsselbyte mit einem Datenbyte per XOR verknüpft:
$crypt(pos, data, key) = data[pos] \ \oplus \ key[pos \ mod \ key_{size}]$
Ein *Secvault* hat also folgende Eigenschaften:

- `id` [0–3]
- `key` (10 Byte)
- `size` [1–1048576]

Die Verwaltung der *Secvaults* soll mit Hilfe des von Ihnen zu entwickelnden Userspace-Tools `svctl` erfolgen:

`USAGE: ./svctl [-c <size>|-k|-e|-d] <secvault id>`

Wird keine Option angebeben, soll die Größe des *Secvaults* in Bytes ausgegeben werden. Die Option `-c` erzeugt einen neuen *Secvault* der Größe `size` Bytes im Kernel. Weiters sollen bei der Option `-c` von *stdin* 10 Zeichen gelesen werden, die als Schlüssel dienen. Der Rest einer längeren Eingabe wird ignoriert; bei einer kürzeren Eingabe wird der Rest des Schlüssels mit 0x0 gefüllt. Die Option `-k` soll die Änderung des Schlüssels eines *Secvaults* erlauben. Dabei soll ebenfalls wie zuvor beschrieben ein Schlüssel von *stdin* eingelesen werden, der den alten Schlüssel ersetzt. Der mögliche Inhalt eines *Secvaults* soll nicht geändert werden. Die Option `-e` löscht die Daten eines existierenden *Secvaults* - d.h. der gesamte Speicher wird Kernel-Modul intern mit 0x0 beschrieben (nicht indirekt über das *Secvault* Character Device). Die Option `-d` soll den *Secvault* aus dem System entfernen und den Speicher wieder freigeben.
Das Kernel Modul legt direkt nach dem Laden (*insmod(8)*) ein eigenes Character Device (ansprechbar über `/dev/sv_ctl`[1] zur Steuerung und Statusabfrage an. Das Userspace-Tool *svctl* soll per *IOCTL* Calls mit dem Kernel Modul kommunizieren.

Anleitung

- Character (und Block) Devices werden über Major und Minor Device Numbers im Filesystem über *Special Files* referenziert. Nehmen Sie 231 als Major Device Number für die *Secvault* Devices. Die *Special Files* können Sie per *mknod(1)* im Verzeichnis `/dev/` anlegen (es empfiehlt sich ein Script dafür im Homedirectory anzulegen).
- Über das *Secvault* Control Device (`/dev/sv_ctl`) können per *open*, *release* und *ioctl* Kommandos an das *Secvault* Device abgesetzt werden.
- Per Userspace-Tool *svctl* sind mit *IOCTL* Calls folgende Operationen möglich:
  - einen *Secvault* anlegen und dabei Grösse und Schlüssel festlegen
  - die Größe eines *Secvaults* abfragen
  - den Schlüssel eines *Secvaults* ändern
  - einen *Secvault* mit 0x0 initialisieren (= Inhalt löschen)
  - einen *Secvault* entfernen (inkl. Speicherfreigabe)

  Bei der Erstellung eines neuen *Secvaults* soll auch ein neues Character Device (ansprechbar über `/dev/sv_data[0-3]`) angelegt werden.
- Über *Secvault* Data Devices (`/dev/sv_data[0-3]`) soll der Zugriff auf den verschlüsselten Speicher per *open*, *release*, *seek*, *read* und *write* erfolgen.
- Implementieren Sie eine geeignete Behandlung von Fehlerfällen:
  - über Speichergröße des *Secvaults* hinaus lesen/schreiben
  - Anlegen eines existierenden *Secvaults*

Target und Entwicklungsumgebung

Da die Entwicklung von Kernel-Modulen aufgrund der Systemnähe bei Fehlern leicht zum Absturz des kompletten Systems führen kann, haben wir eine Entwicklungsumgebung eingerichtet, die den Normal-Betrieb im TI-Lab nicht stört: Im Verzeichnis:

`/opt/osue/uml/`

[1] Sie müssen dieses Devicefile erst (einmalig) anlegen! Siehe Anleitung.

befindet sich eine User Mode Linux (UML) Installation. Der darin enthaltene Linux Kernel (`vmlinux`) wird als normaler User Prozess ausgeführt und kann auf dem Host-System auch nur Operationen durchführen, die Sie als Benutzer im TI-Lab durchführen können. Insbesondere ist es (in den meisten Fällen) nicht möglich aufgrund von Programmierfehler das Hostsystem (TI-Lab Client bzw. den TI-Lab Application Server) zum Abstürzen zu bringen.
Im angegebenen Verzeichnis finden Sie außerdem das Script `start`, welches bei Ausführung eine Instanz des UML Kernels bootet. Beim Bootprozess nutzt der UML Kernel ein Minimal-Debian System[2] (ohne graphischer Benutzeroberfläche) als Root-Image. Bitte ignorieren Sie Fehlermeldungen betreffend des Netzwerkinterfaces. Am Ende des Bootprozesses werden alle virtuellen Konsolen mit 6 Pseudoterminals am Hostsystem verbunden. Die Ausgabe des `start` Skripts bis zum Ende des Bootvorgangs sieht typischerweise so aus:

```
$ /opt/osue/uml/start
 Using swap file: /home/b/.uml.swap
 Using copy-on-write file: /home/b/.uml.cow
...
systemd[1]: Detected virtualization 'uml'.
systemd[1]: Detected architecture 'x86-64'.

Welcome to Debian GNU/Linux 8 (jessie)!
...
[  OK  ] Reached target Login Prompts.
[  OK  ] Reached target Multi-User System.
[  OK  ] Reached target Graphical Interface.
         Starting Update UTMP about System Runlevel Changes...
[  OK  ] Started Update UTMP about System Runlevel Changes.
Virtual console 5 assigned device '/dev/pts/13'
Virtual console 4 assigned device '/dev/pts/15'
Virtual console 3 assigned device '/dev/pts/16'
Virtual console 2 assigned device '/dev/pts/18'
Virtual console 1 assigned device '/dev/pts/19'
Virtual console 6 assigned device '/dev/pts/22'
random: nonblocking pool is initialized
```

Achtung: Das Terminal, in dem Sie die UML Instanz gestartet haben, reagiert nun auf keine Eingaben mehr.
Beachten Sie, dass diese Pseudoterminals je nach Auslastung vergeben werden - d.h. dass Sie i.d.R. nicht immer die gleichen IDs haben. Nun können Sie sich in einem anderen Terminalfenster beispielsweise per `screen` mit einem der Pseudoterminals verbinden (1x Return Taste nach dem Starten von Screen drücken):

```
# screen /dev/pts/25

Debian GNU/Linux 8 sysprog-bonus tty6

sysprog-bonus login:
```

Der Login lautet: `root`
Das Passwort: `rootme`
Im UML System haben Sie nun root-Rechte und können mit der Entwicklung des Kernel-Moduls beginnen. Wir haben Ihnen dafür noch ein Script geschrieben, welches die Kernelquellen und Ihr Homeverzeichnis in das UML-System einbindet. Führen Sie dazu bitte den Befehl im UML-System aus:

[2] http://www.debian.org

`prepare <ti-lab username>`

Sie finden danach unter `/usr/src/linux/` die Kernel-Quellen und unter `/root/homedir/` Ihr Homeverzeichnis.

Compilieren des Testmodules

Hier eine kurze Session die das Compilieren, Laden und Entfernen eines „Hello-World“ Testmodules zeigt (angenommen wird, dass `~/prepare` bereits aufgerufen wurde):

```
root@sysprog-bonus:~# ls /usr/src/linux
COPYING        Makefile        block     init    modules.builtin  sound
CREDITS        Module.symvers  crypto    ipc     modules.order    tools
Documentation  README          drivers   kernel  net              usr
Kbuild         REPORTING-BUGS  firmware  lib     samples          virt
Kconfig        System.map      fs        linux   scripts          vmlinux
MAINTAINERS    arch            include   mm      security         vmlinux.o
root@sysprog-bonus:~# cd ~
root@sysprog-bonus:~# cp test_module homedir/ -R
root@sysprog-bonus:~# cd homedir/test_module
root@sysprog-bonus:~/homedir/test_module# make clean all
make V=1 ARCH=um -C /usr/src/linux M=$PWD clean;
make[1]: Entering directory '/media/host/linux-4.0.2'
[...]
mkdir -p /root/homedir/test_module/.tmp_versions ; rm -f /root/homedir/test_module/.tmp_versions/*
make -f ./scripts/Makefile.build obj=/root/homedir/test_module
[...]
make[1]: Leaving directory '/media/host/linux-4.0.2'
root@sysprog-bonus:~/homedir/test_module# insmod ./tm_main.ko
root@sysprog-bonus:~/homedir/test_module# rmmod ./tm_main.ko
root@sysprog-bonus:~/homedir/test_module# dmesg | tail
[...]
Hello World! I am a simple tm (test module)!
Bye World! tm unloading...
```

Hinweise

- Entwickeln Sie nach Möglichkeit auf einem TI-Lab Client (d.h. ein tiXX.tilab.tuwien.ac.at Rechner). Diese sind - sofern eingeschaltet - ebenfalls per ssh direkt von außen erreichbar.
- Sie können den Quelltext des Kernelmoduls auch am Host editieren (innerhalb ihres Homeverzeichnisses)
- Das Terminallayout können Sie mit dem Kommando `resize` auf die Größe des *screen*-Fensters anpassen. Nach einem Login ist die Größe auf nur 80 Spalten und 25 Zeilen gesetzt.
- Die UML Instanz können Sie sauber terminieren, indem Sie im UML System den Befehl `halt` ausführen.
- Entwickeln und Testen Sie nach Möglichkeit in kurzen Zyklen: Debugging, wie Sie es im Userspace per *gdb* gewohnt sein mögen, steht Ihnen nur mit sehr viel mehr Aufwand im Kernelmode zur Verfügung. Erweitern Sie daher Ihr Modul in kleinen Schritten und testen gründlich bereits implementierte Funktionen, bevor Sie darauf aufbauen.
- Es können die Character Device Files im Filesystem (unter `/dev/`) unabhängig von den tatsächlich vorhandenden Devices im Kernel existieren (d.h. diese müssen nicht direkt vom Modul angelegt werden bzw. wieder gelöscht werden).

- Registrieren Sie gleich beim Laden des Moduls eine *character device region*, in der alle fünf Devices (ein Control- und vier *Secvault*-Devices) Platz finden.
- Sie können das Unixtool *dd(1)* zum Testen der einzelnen *Secvaults* verwenden. Testen Sie insbesondere Ihre Behandlung der möglichen Fehlerfälle (z.B. Schreiben/Lesen über die Grenzen des *Secvaults*, ...).
- Mit dem Kommando *su(1)* können Sie die Identität von einem der drei Testuser (test1, test2 oder test3) annehmen.

Fragen

- Ist es Ihnen möglich den *Secvault* so einzurichten, dass nur diejenigen Benutzer, die ihn angelegt haben, darauf schreiben und davon lesen können? Jeder Benutzer des Systems soll, sofern der angeforderte *Secvault* frei ist, in der Lage sein, einen *Secvault* anzulegen.
  Bemerkung: bis zu +5 weitere Bonuspunkte bei korrekter Implementierung.
- Wie verhindern Sie unsynchronisierten Zugriff bei „gleichzeitiger“ Verwendung des selben *Secvault* Data Devices ?
- Angenommen, ein Angreifer möchte den unverschlüsselten Inhalt eines *Secvaults* kennen und hat die Möglichkeit RAM direkt auszulesen[3]. Andere Systemkomponenten, insb. der *Secvault* Schlüssel, CPU und Programmcode, sind für den Angreifer jedoch nicht einsehbar oder sogar beeinflußbar. Ist der Angreifer in der Lage den Plaintext des *Secvaults* wiederherzustellen? Welche maximale Secvaultgröße wäre Ihrer Meinung nach sicher oder ist jede Größe sicher/unsicher?

Literatur

[1] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman *Linux Device Drivers*, O'Reilly Media, Inc., 3rd Edition, 2005, 0596005903, Verfügbar unter: http://lwn.net/Kernel/LDD3/.

[3] Die Nintendo 3DS Spielekonsole nutzte beispielsweise einen externen RAM Bauteil, wo Daten abgelegt wurden. Ein Angriff auf die CPU Kerne in einem System-on-Chip (SoC) Package sowie auf den Programmcode war aus diversen Gründen unpraktisch. Die Daten in diesem externen RAM Bauteil waren jedoch abgreifbar. Siehe: https://owncloud.tuwien.ac.at/public.php?service=files&t=401f7fe05e214ea6acaeeffabd1224d7 (by Neimod)