Protokoll
BS Praktikum 4
Swaneet Sahoo und Ivan Morozov
7.1.14

Aufgabenbeschreibung:

Zu schreiben ist ein Kerneltreiber welches einem erlaubt Text mit einer einfachen Zeichenersetztung zu verschlüsseln. Die beiden Devices translate0 und translate1 übernehmen diese Funktionen. Das translate0 verschlüsselt Klartext und translate1 entziffert Ciphertext.
Zusätzlich sollen noch ein Script zum Deinstallieren und Installieren verwendet werden. Weiterhin ist der Verschlüsselungsstring und die Buffersize beim Laden des Moduls einstellbar.

Entwurf des Moduls translate:

(Der Entwurf ist nicht viel anders als im vorher angegebenen Entwurfsdokument. Wir haben uns am dritten Kapitel wo scull erklärte wurde orientiet.)
Das Translate-Modul wurde ähnlich dem scull-Beispiel entwickelt. Im Header wird das struct translate_dev definiert, welches diverse Pointer für die Buffer sowie das struct cdev und zwei Semaphoren für den Lese-/Schreibzugriff hat. Im Header werden außer den Funktionsdeklarationen auch die Device-Fileoperations implementiert(bzw. wird dort auf die einzelnen Funktionen verwiesen.) Im Header stehen außerden noch globale Konstanten und hilfreiche Makros die z.B. für die Kodierung/Dekodierung verwenden werden.

Das translate_open führt je nach Write/Read-modus auf dem dazugehörigen Lese-/Schreibsemaphoren ein down_trylock aus um Zugriff auf das Device zu haben. Wenn bereits ein anderer Prozess vom Device liest/schreibt, kann nun kkein anderer Prozess dazukommen. (Es ist aber möglich, dass ein Prozess vom Device lies und ein anderer auf dem gleichen Device schreibt.) Wir haben den Semaphroren auf den Wert 1 initialisiert, damit nur jeweils ein Prozess zugriff hat.
Das Translate_close gibt einfach den entsprechenden Semaphroren frei(bzw. veringert die Anzahl der lesenden/schreibenden Prozesse).
Im translate_write werden nach und nach die Chars aus dem Userspace kopiert. Falls es sich um das translate0-Device handelt, wird es außerdem noch verschlüsselt. Ist zwischendurch der Buffer voll, wird vorzeitig abgebrochen (und die Anzahl der bisher übergebenen Chars zurückgegeben).
Im translate_read passiert das genaue Gegenteil. Es wird dort Char für Char ins Userspace kopiert und im Falle der translate1 dekodiert. Ist zwischendurch der Buffer leer, wird vorzeitig abgebrochen.

Bei der Kodierung wird anhand der ASCII-Zahl des Buchstabens ermittelt, welcher Buchstabe aus dem translate_substr zu verwenden(encode_index_from_char). Die ersten n/2 Buchstben geben die Kodierung für Kleinbuchstaben an. (n steht für die Länge des translate_substr). Die anderen Buchstaben geben die Kodierung für Großbuchstaben an.
Für das Dekodieren wird aus die mit strchar() berechnete Position des Chars im im translate_substr die ASCII Zahl des kodierten Buchstabens berechnet(decode_from_index). Diese ASCII-Zahl wird dann direkt als Char wieder geschrieben. Falls der Cipher nicht im substr gefunden wurde, dann war es kein verschlüsselter Char. Dann wird es einfach unverändert gelassen.

Das translate_cleanup, translate_init und translate_setup_cdev wurden aus scull übernommen. Dort werden die Werte des translate_dev struct initialisiert/zurückgesetzt und der Speicher für die Devices und die Buffer werden alloziert oder freigegeben.

Das Makefile ist fast identisch mit dem aus dem scull. Zwei Scripte install.sh und uninstall.sh übernehmen das (De-)Installieren des Moduls. Das Install.sh macht das gleiche wie der Script für die scull-Installation aus dem dritten Kapitel aber führt davor noch die Kompilierung aus. Das uninstall.sh ruft einfach ein rmmod auf, entfernt die Device-Nodes und löscht alle Kompilationsdateien.

Hinweis:

Wir hatten zunächst versucht ein einfaches Hello-Kernelmodul zu installieren(wie sie es in der Vorlesung vorgestellt hatten.) Die Kompilation ist jedoch fehlgeschlagen, da in dem Ordner /modules/ 3.7.10-1.1-default der /build Ordner fehlte. Nach etwas Recherche stellte sich herraus, dass openSuse diverse Kernel Headers fehlten. Wir haben dann mit „$ sudo zypper in -t pattern devel_kernel“ die Headers installiet. Nach einem Neustart war im oben angegebenen Ordner leider immernoch kein build Ordner und die Compilation schlug immernoch fehl. Die Installation war hat aber einige andere Ordner in /modules hinzugefügt.

```
$ ls
3.7.10-1.1-default   3.7.10-1.24-desktop  3.7.10-1.24-xen
3.7.10-1.24-default  3.7.10-1.24-pae      3.7.7-1.2-default
```

Die Ordner die auf .24 enden kamen dazu. In dem ...24-default Ordner ist ein /build und /source Ordner zu finden.
Wir haben dann zum Kompillieren den Pfad im Makefile geändert:

```
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
```

Zu

```
KERNELDIR ?= /lib/modules/3.7.10-1.24-default/build
```

Mit diesem Pfad funktionierte das Kpmpilieren und das installieren. Wir haben uns dann auf die eigentliche Aufgabe konzentriert, anstatt noch diesen Fall allgemein zu halten.
Nach ihrem Input dazhu, ist uns klar, dass dies funktioniert ein großes Glück ist.

Ergebnisse:

Mit subst="zyxwvutsrqponmlkjihgfedcbaZYXWVUTSRQPONMLKJIHGFEDBCA":

```
$ echo "We're all some1 else to some1 else." > /dev/translate0
$ cat /dev/translate0 > /dev/translate1
$ cat /dev/translate1
We're all some1 else to some1 else.
$ echo "We're all some1 else to some1 else." > /dev/translate0
$ cat /dev/translate0
Dv'iv zoo hlnv1 vohv gl hlnv1 vohv.
```

Ohne uebergebenen substr-Parameter: (Dann werden Groß und Klein vertauscht.)

```
$ echo "We're all some1 else to some1 else." > /dev/translate0
$ cat /dev/translate0
wE'RE ALL SOME1 ELSE TO SOME1 ELSE.
```

Verhalten bei langen Eingaben:
Bash 1:

```
$ echo "… langer String …" > /dev/translate0
```

(bash 1 hängt jetzt)
In bash 2:

```
$ cat /dev/translate0 > /dev/translate1
$
```

Nun hängt bash 1 nicht mehr.
Jetzt kann man in einem beliebigen bash das Ergebnis ausgeben.

```
$ cat /dev/translate1
```