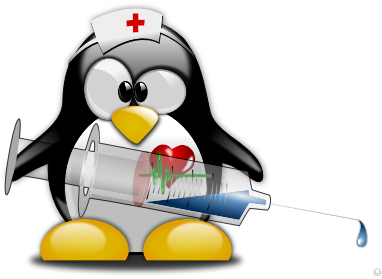
Fachprojekt „Systemsoftwaretechnik“
A2 – Debugging im Linux-Kern

2 Debuging im Linux-Kern - Teil 2



In dieser Aufgabe befasst Ihr Euch mit den externen Werkzeugen zur Fehlersuche in Linux. Hierzu erhaltet Ihr von uns einen präparierten Linux-Kern, den Ihr passend übersetzen müsst. Ihr befasst Euch mit den in der Übung vorgestellten Werkzeugen und lernt sie anzuwenden.

Ladet Euch hierzu mittels `git` den Quellcode für die Aufgaben aus der Vorgabe[1] herunter, sofern Ihr dies noch nicht für den vorherigen Zettel getan habt. Achtet auf die Hinweise **am Ende** des Übungsblatts.

2.1 Kernel-Debugger

In dieser Aufgabe untersucht Ihr die Abläufe im Inneren des Linux-Kerns mit einem Debugger, hier dem `GDB`. Im Internet gibt es verschiedene Befehlsreferenzen[2]. Ihr könnt außerdem auf das GDB-Tutorial der Veranstaltung „Betriebssystembau“[3] zurückgreifen.

Aktualisiert bitte Eure Version des Linux-Kerns (siehe `git pull`). Anschließend wechselt auf den Zweig `sst` und nutzt die Konfiguration `sst-base.config` als Basis für die Aufgabe. Aktiviert nun die Optionen für den Kernel-Debugger `KGDB`. Achtet darauf, dass die Unteroption für die Verwendung des Kernel-Debuggers über eine serielle Konsole aktiviert ist.

Bevor Ihr nun die virtuelle Maschine startet, beantwortet folgende Fragen:

a. Welche Kern-Parameter müssen übergeben werden, damit Ihr a) den Kernel-Debugger über die zweite Serielle erreicht und b) der Linux-Kern beim Starten die Ausführung unterbricht, um auf den GDB zu warten?
Hiermit ist *nicht* das Warten des Emulators (QEMU) auf die Verbindung vom GDB gemeint.

b. Wofür steht der Kern-Parameter `nokaslr`? Welchen Zweck hat die Funktionalität, die hier abgeschaltet wird?

2.1.1 Anwendung des `kgdb`

Startet nun das `boot.sh`-Skript mit dem zusätzlichen Parameter `-s`[4] und im Anschluß den GDB mit dem Pfad zu der Datei `vmlinux` sowie `-ex "target remote localhost:1234"` als Kommandozeilenparameter. Beobachtet nun Eure virtuelle Maschine sowie die Konsole, in der Ihr den GDB gestartet habt. Nach einer bestimmten Zeit sollte der Linux-Kern den Startprozess anhalten, der GDB eine Eingabeaufforderung starten und eine Ausgabe tätigen.

Hinweis: Übergebt Eurem Kern unbedingt noch diese Parameter `nokaslr rodata=off dyndbg=\" module sst_chrdev +p;module sst_common +p;\"`

a. Bestimmt mit Hilfe des GDBs die Aufrufhierarchie. Wie lautet die erste **C**-Funktion am Anfang der Aufrufhierarchie?

b. Wo im Quellcode (Datei + Zeile) liegt die Funktion, die gerade unterbrochen wurde?

[1] ssh://mars.cs.tu-dortmund.de/fs/scratch/al/sst-linux
[2] https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf
[3] https://sys-sideshow.cs.tu-dortmund.de/lehre/ws23/bsb/folien/seminar-gdb.pdf
[4] Kopiert Euch ggf. eine aktuelle Version von `/fs/proj/sst/`

2.1.2 Unterbrechen von Abläufen im Kern

Haltet in Stichpunkten fest, wie Ihr folgende Aufgaben manuell mittels `kgdb` umgesetzt habt:

a. Beim Schreiben in `/dev/the-universe` soll eine Frage in eine Aussage umgewandelt werden.

b. Beim Lesen aus /dev/the-universe soll die Antwort in eine Frage umgewandelt werden.

c. Setzt Breakpoints auf die Funktionen `sst_produce_answer()` sowie `sst_produce_question()`. Was ist die oberste Funktion der Aufrufhierarchie **nach** den jeweils unterbrochenen `sst`-Funktionen?

d. Wechselt mit dem GDB innerhalb dieser Funktionen einen Stapelrahmen (engl. `Stackframe`) nach unten und schaut Euch in der jeweils aufrufenden Funktion um. Welche lokalen Variablen seht Ihr?

e. *Optionale Aufgabe:* Modifiziert beim Lesen aus dem Universum die **komplette** Antwort, so dass „void.“geantwortet wird. Achtet dabei auf die Länge der Zeichenkette `answer`.

Hinweis: Die Veränderungen sollen nicht automatisch bei jeder Ausführung der unterbrochenen Funktionen ablaufen. Ihr sollt nur festhalten, welche Schritte Ihr mit dem `kgdb` bei einer einmaligen Unterbrechung ausführen müsst.

2.2 SystemTap

Nun sollt Ihr Euch mit dem Instrumentierungswerkzeug `SystemTap` beschäftigen. Doch bevor Ihr starten könnt, müsst Ihr Veränderungen an Eurem Gast-Betriebssystem vornehmen:

- Fügt in die Datei `/etc/apt/sources.list` folgende Zeile ein:
  `deb http://deb.debian.org/debian-debug stable-debug main`
- Führt eine Aktualiserung der Paketliste durch: `apt update`.
- Installiert folgende Pakete: `systemtap build-essential tshark tcpdump libssl3-dbgsym curl`.
- Verschiebt Eurer Quellcodeverzeichnis des Linux-Kerns (meistens `linux/`) in das Verzeichnis `vmshare/`, damit die virtuelle Maschine über `9P` darauf zugreifen kann.

Achtung: Das Host- **und** Gastsystem müssen die gleiche Compiler- und libc-Version nutzen. Am Besten sollten sie die gleiche Betriebssystemversion verwenden. Beispielsweise Debian 12 Bookworm. Auf den Laborrechnern sowie mars ist Debian 12 bereits installiert.

Achtung: Es gibt in Debian 12 ein Bug in `SystemTap`. Verwendet die `SystemTap`-Installation auf den Poolrechnern, wenn Ihr Eure Skripte dort übersetzt: `/fs/proj/sst/stap/bin/stap`. Ansonsten führt in Euren virtuellen Maschine folgende Schritte aus:

```
1 sst@sst: $ wget https://sys-sideshow.cs.tu-dortmund.de/lehre/ss25/sst/stap-ak.tar.xz
2 sst@sst: $ sudo tar -C / -xJf stap-ak.tar.xz
3 sst@sst: $ /opt/stap-ak/bin/stap
```

Übernehmt die Konfiguration des Linux-Kerns aus Aufgabe 2.1. Aktiviert jedoch zusätzlich die Option `Kernel Function Tracer` sowie alle vier Unteroptionen. Außerdem müsst Ihr die Modulunterstützung im Hauptmenü, die Option `uprobes-based dynamic events` aktivieren und die Option `Kprobes`. Führt nun folgende Aufgaben mit `SystemTap` durch und denkt an die Dokumentation zu den von `SystemTap` bereitgestellten Funktionen (`Tapsets`[5]).

[5] https://sourceware.org/systemtap/tapsets/

2.2.1 Das erste SystemTap-Skript

Schreibt nun ein SystemTap-Skript, die Länge der Strings bei Aufrufen der Funktion `sst_produce_answer()` speichert und am Ende des Skripts als Histogram ausgibt. Übergebt `stap` noch die folgenden Argumente: `-v -r /XX/linux`. Letzteres entspricht dem Pfad innerhalb der virtuellen Maschine zu Eurem Quellcodeverzeichnis des Kerns. Probiert auch gerne einmal den Paramter `-t` aus. Er sorgt dafür, dass am Ende der Ausführung eine Zusammenfassung der ausgeführten Probepoints sowie der Timinginformationen ausgegeben wird. SystemTap schreibt alle Ausgaben anschließend auf die Konsole[6]. Um das Programm anzuhalten, müsst Ihr `Strg+C` verwenden.

2.2.2 SystemTap statt kgdb

Realisiert nun die Teilaufgaben a und b aus Abschnitt 2.1.2 mittels SystemTap. Natürlich werden die Schritte bei *jedem* Lesen bzw. Schreiben durchgeführt. Als *optionale Aufgabe* dürft Ihr gerne die Aufgabe e mit SystemTap umsetzen.

2.2.3 Pssst! Geheim!

Ladet Euch das Skript `capture-ssl.stp`[7] herunter. Allerdings fehlt in dem Skript noch die Größe des Buffers, in dem der Schlüssel abgelegt ist – siehe `XX` in dem Skript. Diesen müsst Ihr noch ergänzen. Schaut Euch hierzu den Quellcode[8] der Bibliothek `libssl` an, sucht nach der instrumentierten Funktion und lest die Größe des Schlüssels aus dem Anwendungsprozess aus. Schaut hierzu auch in die Dokumenation von SystemTap, wie Ihr einen Zahlenwert aus dem Nutzerprozess auslesen könnt. Führt anschließend die folgenden Schritte aus. Ggf. müsst Ihr mehrere SSH-Verbindungen aufbauen, um alle Programme zu starten.

```
1 sst@sst: $ tcpdump -s0 -w /mnt/traffic.pcap port 443
2 sst@sst: $ stap -r /mnt/linux/ -g ./capture-ssl.stp | tee keylog.txt
3 sst@sst: $ curl --tls-max 1.2 -o /dev/null ↙
    https://sys-sideshow.cs.tu-dortmund.de/lehre/ss25/sst/ein-text.txt
4 sst@sst: $ tshark -o tls.keylog_file:keylog.txt  -x -r traffic.pcap -V
```

Ladet nun, wie oben angegeben, eine Datei von unserem Webserver. Beendet nun `stap` und `tcpdump`. Ruf abschließend `tshark` auf, um den Datenverkehr zuentschlüsseln: Welcher Inhalt steht in der heruntergeladenen Datei?

Wofür müsst Ihr das Paket `libssl3-dbgsym` installieren?

[6] Schaut in die Handbuchseiten zu SystemTap, um die Ausgaben in eine Datei zu schreiben.
[7] https://sys-sideshow.cs.tu-dortmund.de/lehre/ss25/sst/capture-ssl.stp
[8] https://github.com/openssl/openssl

Allgemeine Hinweise:

- Falls Ihr der Aufruf von `stap` zu lange dauert, könnt Ihr den Vorgang aufsplitten:
  a. Erst das Modul auf dem Host (z. B. Labor-Rechner) übersetzen: `stap -v -p 4 -r /XXX/linux -m meinmodul.ko YY/modul.stp`.
  Der Pfad zu eurem `linux` *muss* ein *absoluter* Pfad sein. Außerdem muss das resultierende Kern-Modul für die VM zugreifbar sein. Lasst es z. B. direkt in Eurem `vmshare`-Verzeichnis erstellen.
  b. Startet anschließend *in* der `VM` `staprun`: `staprun meinmodul.ko`
- Falls der Bildschirm nicht ausreicht, um eine Fehlermeldung vollständig zu betrachten, könnt Ihr die serielle Konsole nutzen. Hierzu müsst Ihr dem Linux-Kern einen zusätzlichen Parameter übergeben[a]. Schaut Euch neben der Dokumentation auch die Hilfeausgabe des Skripts `boot.sh` an. Diese zeigt Euch, wie Ihr Parameter an den Kern übergeben könnt.
- Das `boot.sh`-Skript leitet den Netzwerkport 22 an den Port `4711` des Hosts weiter. So könnt Ihr Euch, sofern SSH installiert ist, via SSH in der `VM` anmelden. Ihr müsst dem `ssh`-Kommando nur die richtigen Parameter Port angeben – siehe `man ssh`. Verwendet als Host `localhost`. Wenn Ihr außerdem in `/etc/ssh/sshd_config` die Variable `PermitRootLogin` einkommentiert und auf `yes` setzt, könnt Ihr Euch direkt als `root` per SSH anmelden.

[a] https://www.kernel.org/doc/html/latest/admin-guide/serial-console.html