Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 99

Kern-Technik

Nach einem Kernel-Crash sammelt Linux noch im Todeskampf emsig Informationen zum Tathergang. Nur – wohin mit dem Vermächtnis, wenn schon alles verwüstet ist? Das virtuelle Filesystem Pstore hat als letzter Begleiter stets ein offenes Ohr. Eva-Katharina Kunst, Jürgen Quade

© psdesign1, Fotolia

Starman (links in Abbildung 1) ist in einem Elektro-Gebrauchtwagen von Elon Musk unterwegs im Sonnensystem. Auf seiner vermutlich mehrere Millionen Jahre dauernden Reise hat er im Dashboard des Tesla-Roadsters ständig die Nachricht „Don't Panic" vor Augen (Bildmitte). Urheber dieser mystischen Meldung ist Douglas Adams, der den guten Rat in großen, freundlichen Buchstaben auf die Umschlagseite des legendären Reiseführers „Per Anhalter durch die Galaxis" hat setzen lassen.

Die Autoren

Eva-Katharina Kunst ist seit den Anfängen von Linux Fan von Open Source. Jürgen Quade ist Professor an der Hochschule Niederrhein. Ihr gemeinsames Buch „Linux-Treiber entwickeln" ist Ende 2015 in vierter Auflage erschienen.

„Don't Panic" lässt sich nicht nur als guter Rat an Starman oder den Romanhelden Arthur Dent deuten, sondern auch als Bitte computerkundiger Tesla-Fahrer an die Linux-basierte Fahrzeugsteuerung.

Denn Unix-Systeme enthalten schon immer, also bald 50 Jahre lang, eine Funktion namens »panic()«. Die Funktion wird in aussichtslosen Situationen aufgerufen und hat neben einer Fehlermeldung den Komplettausfall des Betriebssystems zur Folge. Im aktuellen Linux-Kernel finden sich fast 600 Stellen, die bereit sind eine »panic()« auszulösen.

Ein letzter Gruß

Der Kernelprogrammierer gibt der Funktion »panic()« noch eine Abschiedsbotschaft mit, bevor auf der Konsole ein Wust für die meisten Anwender unverständlicher Zeichen und Symbole den Abgang begleitet. Bei genauerer Hinsicht erkennen Entwickler den Call- respektive Stack-Trace und die Inhalte aller CPU-Register zum Zeitpunkt des Exitus.

Das Vermächtnis umfasst durchaus wertvolle Information, wenn es um die Aufklärung der Todesursache geht – ein Protokoll der letzten Aktivitäten. Es zeigt ganz oben an, welche der rund 600 Funktionen die »panic()« ausgelöst hat, wer diese Funktion aufgerufen hat und von welcher Funktion sie dann aufgerufen wurde. Diese Aufrufkette ist bis zum Anfang verfolgbar.

Untermauert mit den Speicheradressen ist sogar erkennbar, wo genau innerhalb einer Funktion die nächste Funktion aufgerufen wird, denn immerhin könnte es mehrere potenzielle Stellen geben. Dies wird allerdings nur auf Maschinencode-Ebene beziehungsweise über Speicheradressen erkennbar. Zur Dekodierung sind demnach der Maschinencode und auch ein Disassembler hilfreich.

Das Hauptproblem bei einer Panic ist allerdings, dass die Maschine beim Crash nicht mehr viele Möglichkeiten besitzt, der Welt ihre Informationen mitzuteilen. Grafischer Firlefanz ist sowieso nicht zu erwarten, denn das Netzwerk und die Festplattenzugriffe sind schon in Mitleidenschaft gezogen.

Daher schickt Linux seinen Output zur Konsole, die seit den Unix-Anfangstagen direkt mit dem eigentlichen Rechner über eine serielle Schnittstelle verbunden ist (Abbildung 2). In Zeiten der Virtualisierung und des Cloud Computings existiert aber in der Regel keine physisch zugängliche Konsole mehr.

Die Überlebenden

Der glückliche Admin, der eine Maschine mit persistentem Hauptspeicher besitzt, kann sich allerdings mit Pstore und Ramoops behelfen (Abbildung 3). Persistenter Speicher behält seinen Inhalt auch nach einem Kappen vom Stromnetz. Falls beim Neustart der Hauptspeicher nicht gelöscht wird, lässt sich der Speicher nutzen, um Informationen zwischenzuparken. Mit Hilfe von Pstore und des zuge-

hörigen Ramoops schreibt dann »panic()« seine Daten nicht nur auf die Konsole, sondern zusätzlich in einen definierten Bereich des Hauptspeichers, den Linux ansonsten ausspart.

Der Bereich ist begrenzt und als Ringpuffer organisiert. Das virtuelle Filesystem Pstore macht sowohl die Schreibzugriffe für den Kernel als auch das spätere Auslesen für Applikationen sehr einfach. Pstore präsentiert nämlich die von »panic()« abgelegten Daten als Dateien, die der Administrator mit Werkzeugen wie »cat« auslesen darf (Abbildung 4). Wer die Dateien per »rm« löscht, gibt den Speicher wieder frei für das nächste Desaster-Log.

Pstore ermöglicht wie erwähnt den Zugriff über Dateien sowohl aus dem Kernel als auch aus dem Userland. Dabei liegt die Betonung auf dem Plural: Dateien. So ist Ramoops – als Teil von Pstore – nicht der einzige Nutzer. Falls entsprechend konfiguriert, lassen sich Daten auf der Konsole dort speichern oder auch Daten von Ftrace [1]. Ubuntu und ebenso Raspbian binden das Pstore-Dateisystem übrigens beim Booten automatisch ein. Wer Gleiches von Hand macht, kann die Größe als Option angeben:

```
mount -t pstore -o kmsg_bytes=8000 none➥
 /sys/fs/pstore
```

Ramoops selbst bietet eine ganze Reihe von Konfigurationsoptionen. Es fängt bei der Adresse und der Größe des Hauptspeicherbereichs an und reicht bis zur Aufgabenstellung, nur Panik-Meldungen oder auch Oops-Meldungen abzulegen.

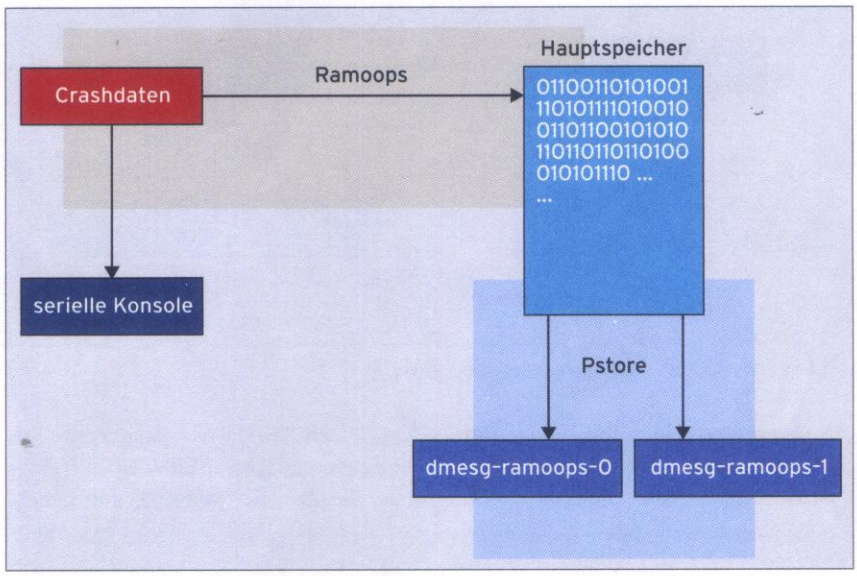

Abbildung 2: Der Klassiker: Der Kernel schickt die Crashdaten auf eine angeschlossene Konsole.

Der Systemverantwortliche konfiguriert dies alles beim Laden des Kernelmoduls per Übergabeparameter, alternativ auch über das Sys-Filesystem. Letzteres ermöglicht per Lesezugriffe zudem das Steuern der dort abgelegten Werte (Listing 1). Wer Hardware verwendet, die über einen Devicetree organisiert ist, etwa einen Raspberry Pi, bekommt die Parameter auch auf diesem Wege ins System.

Vorarbeiten

Wer im Besitz eines Raspberry Pi ist und sich nicht scheut, selbst einen Kernel zu generieren, kann Pstore und Ramoops mit wenigen Handgriffen ausprobieren. Zwar besitzt das Beerchen keinen persistenten Hauptspeicher, solange aber sein Hauptspeicher Strom bekommt, bleibt auch dessen Inhalt erhalten – meist reicht das bereits.

Der eigene Kernel wird nötig, weil Pstore und die zugehörige Funktionalität Ramoops auf gängigen Raspberry-Linuxen fehlen. Unter [2] findet sich eine gute Backanleitung für einen Pi-Kernel. Wer es eilig hat, wählt die Variante „Cross-Generierung auf dem PC". Etwas einfacher – wenngleich zeitaufwändiger – ist dagegen der Weg, Kernel und den benötigten Devicetree-Overlayblob auf der Beere selbst zu generieren.

Für diesen Weg – den Kernel auf der Minimaschine selbst kompilieren – sind zuvor noch ein paar Prögrämmchen wie Bison, Flex, Bc und Libssl-dev per »apt-get« ins System zu holen (Listing 2). Anschließend installiert »git clone« den himbeerspezifischen Linux-Quellcode. Hier ergibt sich übrigens auch eine

© Space X (Public Domain)

Abbildung 1: Der Elektroschrott-Presse entkommen: Space-X-Gründer Elon Musk ließ als Nutzlast bei der Falcon Heavy Demo Mission einen seiner Tesla-Roadster ins All schießen. Dass sich im Raumanzug am Steuer ein Tesla-Angestellter auf der Abschussliste (Starman) so seiner Entlassung entzogen hat, ist Spekulation.

Listing 1: Ramoops sind umfangreich konfigurierbar

```
01 root@raspberrypi:~# grep "" /sys/module/ramoops/
   parameters/*
02 /sys/module/ramoops/parameters/console_size:0
03 /sys/module/ramoops/parameters/dump_oops:1
04 /sys/module/ramoops/parameters/ecc:0
05 /sys/module/ramoops/parameters/ftrace_size:0
06 /sys/module/ramoops/parameters/mem_address:184549376
07 /sys/module/ramoops/parameters/mem_size:65536
08 /sys/module/ramoops/parameters/mem_type:0
09 /sys/module/ramoops/parameters/pmsg_size:0
10 /sys/module/ramoops/parameters/record_size:16384
```

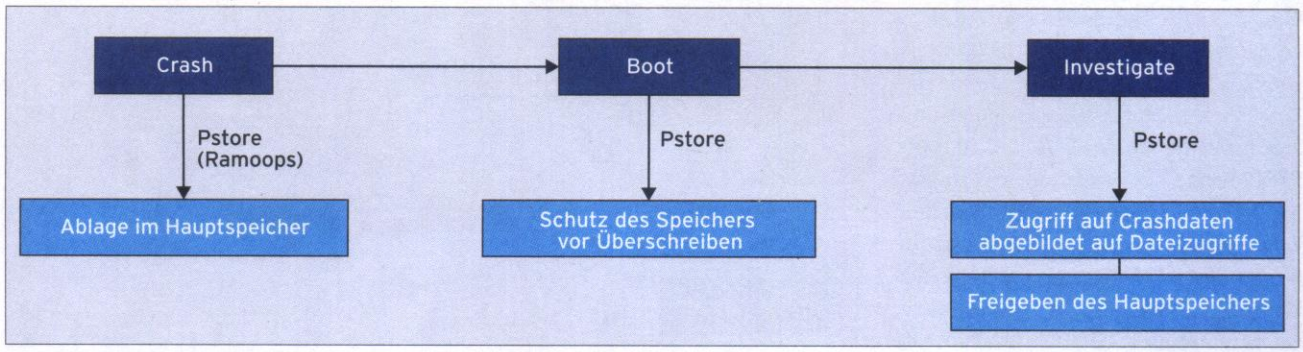

Abbildung 3: Die Aufgaben des virtuellen Filesystems Pstore.

gute Gelegenheit, einen aktuellen Kernel, zum Beispiel 4.17, per »git checkout rpi-4.17.y« zu wählen.

Um einen zur vorliegenden Hardware passenden Betriebssystemkern zu erzeugen, setzen Raspberry-1-Besitzer die Environmentvariable »KERNEL« auf »kernel«, Raspberry-2- und -3-Eigner benutzen »kernel7«. Danach geht's ans Vorkonfigurieren des Kernels: Per »make menuconfig« gestartet, steht die Einstellung für Pstore im Menü unter »File systems | Miscellaneous filesystems | Persistent store support« bereit.

Das zweimalige Betätigen der [Leer]-Taste bindet das Subsystem in den Kernel als festen Bestandteil ein, statt als ladbares Modul, was am Stern (»*«) sichtbar wird (**Abbildung 5**). Wichtig ist an dieser Stelle, auch »Log panic/oops to a RAM buffer«, also Ramoops, auszuwählen.

```
pi@raspberrypi:~ $ sudo su
root@raspberrypi:/home/pi# cd /sys/fs/pstore/
root@raspberrypi:/sys/fs/pstore# ls -l
total 0
-r--r--r-- 1 root root 23115 Jul  6 08:03 dmesg-ramoops-0
-r--r--r-- 1 root root 27264 Jul  6 08:03 dmesg-ramoops-1
root@raspberrypi:/sys/fs/pstore# head dmesg-ramoops-0
Oops#1 Part1
<6>[    0.000000] Booting Linux on physical CPU 0xf00
<5>[    0.000000] Linux version 4.17.3-v7+ (root@raspberrypi) (gcc version 6.3.0
 20170516 (Raspbian 6.3.0-18+rpi1+deb9u1)) #1 SMP Wed Jul 4 20:03:43 UTC 2018
<6>[    0.000000] CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387
d
<6>[    0.000000] CPU: div instructions available: patching division code
<6>[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruc
tion cache
<6>[    0.000000] OF: fdt: Machine model: Raspberry Pi 2 Model B Rev 1.1
<6>[    0.000000] Memory policy: Data cache writealloc
<6>[    0.000000] cma: Reserved 8 MiB at 0x3ac00000
<7>[    0.000000] On node 0 totalpages: 242688
root@raspberrypi:/sys/fs/pstore#
```

Abbildung 4: Nach einer Linux-Panik lassen sich die Crashdaten in Form von Dateien auslesen.

Overlay unterbringen

Außerdem muss der Admin den Devicetree um die Ramoops-Parameter ergänzen. Dazu setzt er den Code aus **Listing 3** in eine Datei »ramoops-overlay.dts« ein und kopiert diese in das Verzeichnis »arch/arm/boot/dts/overlays/«. Damit später aus dem beschreibenden Text das

Listing 2: Ramoops auf Raspberry Pi 2 installieren

```
# Vorbereitung - Software installiern
apt-get install ncurses-dev bison flex bc libssl-dev
cd /usr/src/

# Linux-Quellcode installieren
git clone https://github.com/raspberrypi/linux.git
cd linux

# Aktuelle Kernelversion auswählen
git checkout rpi-4.17.y

# Generierung für Raspberry Pi 2 oder 3
KERNEL=kernel7

# Grundkonfiguration
make bcm2709_defconfig

# Pstore mit Ramoops auswählen
make menuconfig
    [...]

# Mit einem Editor (z.B. Vim) die Devicetree-Datei anlegen
vi arch/arm/boot/dts/overlays/ramoops-overlay.dts
    [...]

# Mit einem Editor (z.B. Vim) das Makefile ergänzen
vi arch/arm/boot/dts/overlays/Makefile
    [...]

# Kernel und Devicetree generieren
make -j4 zImage dtbs

# Devicetreeblob installieren
cp arch/arm/boot/dts/overlays/ramoops.dtbo /boot/overlays/

# Originalkernel retten
cp /boot/kernel7.img /boot/kernel7.img.org

# Neuen Kernel installieren
cp arch/arm/boot/zImage /boot/kernel7.img

# Das Laden des Ramoops-Overlays erzwingen
echo "dtoverlay=ramoops" >>/boot/config.txt
```

Overlay »ramoops.dtbo« entsteht, muss er das Overlay-Makefile um einen Einzeiler ergänzen (Abbildung 6).

Ein »make -j4 zImage dtbs« startet den Übersetzungslauf. Auf einem Raspberry Pi dauert der ziemlich genau so lange wie das Zubereiten einer Tasse Tee. Für einen kurzen Ramoops-Test ist damit schon alles Notwendige vorhanden. Um das System aber produktiv nutzen zu können, bedarf es auch der Module; »make modules« erzeugt und »make modules_install« installiert sie. Dieser Vorgang währt allerdings rund zwei Stunden, also ziemlich genau so lange wie eine japanische Teezeremonie.

Zur Installation schiebt der Systemverantwortliche den neuen Kernel ins Bootverzeichnis unter dem Namen »kernel7.img« (für einen Raspberry Pi 2 oder 3). Der neue Devicetree-Overlayblob »ramoops.dtbo« kommt in das Overlay-Verzeichnis. Damit Linux den Devicetree-Overlayblob lädt, ist in »/boot/config.txt« noch die Zeile »dtoverlay = ramoops« zu ergänzen. Das war's. Die notwendigen Befehle sind in Listing 2 ab Zeile 33 aufgeführt.

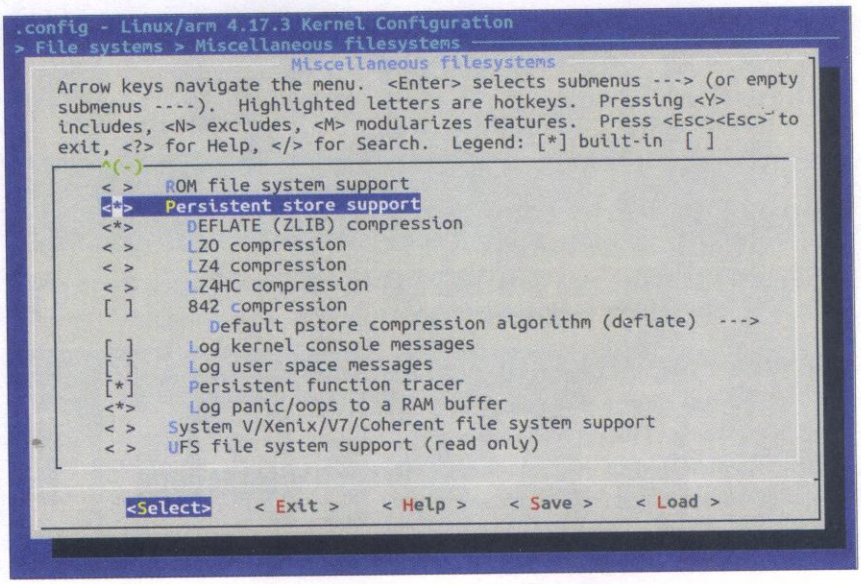

Abbildung 5: Die Kernelkonfiguration zur Konservierung der Crashmeldungen.

Ordentlich crashen lassen

Ein Reboot beweist hoffentlich, dass weiterhin alles funktioniert. Um die neue Funktion zu testen, versetzt der Admin Linux gleich in Panik. Die Kernelfunktion »panic()« sorgt übrigens nicht nur für den Stillstand, sondern organisiert auch die Wiedergeburt. Falls der Admin in die Datei »/proc/sys/kernel/panic« einen (Integer-)Wert schreibt, interpretiert Linux diese Zahl als Verweilzeit in Sekunden bis zur Wiedergeburt, sprich bis zum automatisierten Reboot des Systems:

```
echo 3 >/proc/sys/kernel/panic
```

Bevor der Crash-Pilot mit seinem zerstörerischen Tun beginnt, sorgt er unbedingt dafür, dass die Daten auf dem Hintergrundspeicher in einen leidlich konsistenten Zustand kommen. Dazu gibt er in einer Konsole zunächst »sync« ein. Das Schreiben eines »c« auf die Datei »/proc/sysrq-trigger« lässt den Kernel schließlich abstürzen:

```
sync
echo c >/proc/sysrq-trigger
```

Nach dem Reboot – bitte nicht den Strom abschalten – bietet das Pstore-Subsystem im Verzeichnis »/sys/fs/pstore« die Dateien an, die die Crashdaten enthalten (Abbildung 4). Am besten kopiert man die Dateien ins eigene Homeverzeichnis, um sie daraufhin im Pstore-Verzeichnis zu löschen und Platz für die nächsten Crashdumps zu schaffen. Danach geht es bei einem realen Crash in Ruhe an die Auswertung.

Nur kurz den Blues

Linux wäre nicht Linux, wenn es – wie noch in den 80er und 90er Jahren – auf jeden Nullpointer-Zugriff mit Panik und Tod reagierte. Heute bemerkt der Benutzer nach einer solchen Zumutung allenfalls einen kleinen Schluckauf, ein so genanntes Oops. In dieser Zeit meldet

Listing 3: Devicetree-Overlay für Ramoops [3]

```
01 /dts-v1/;
02 /plugin/;
03
04 / {
05       compatible = "brcm,bcm2835";
06
07       fragment@0 {
08             target-path = "/";
09             __overlay__ {
10                   reserved-memory {
11                         #address-cells = <1>;
12                         #size-cells = <1>;
13                         ranges;
14
15                         ramoops: ramoops@0b000000 {
16                               compatible = "ramoops";
17                               reg = <0x0b000000 0x10000>; /* 64kB */
18                               record-size = <0x4000>; /* 16kB */
19                               console-size = <0>; /* disabled by
   default */
20                         };
21                   };
22             };
23       };
24
25       __overrides__ {
26             total-size = <&ramoops>,"reg:4";
27             record-size = <&ramoops>,"record-size:0";
28             console-size = <&ramoops>,"console-size:0";
29       };
30 };
```

```
root@raspberrypi:/usr/src/linux/arch/arm/boot/dts/overlays# head Makefile
# Overlays for the Raspberry Pi platform

dtbo-$(CONFIG_ARCH_BCM2835) += \
        ramoops.dtbo \
        adau1977-adc.dtbo \
        adau7002-simple.dtbo \
        ads1015.dtbo \
        ads1115.dtbo \
        ads7846.dtbo \
        akkordion-iqdacplus.dtbo \
root@raspberrypi:/usr/src/linux/arch/arm/boot/dts/overlays#
```

Abbildung 6: Im Makefile ist eine Ergänzung notwendig.

das System den illegalen Zugriff und geht dann lässig seiner göttlichen Mission weiter nach – Jake und Elwood Blues lassen grüßen. Es obliegt dem Admin, durch einen Reboot das System wieder in vollständige Konsistenz zu versetzen.

Der Vorteil der Überlebensstrategie besteht darin, dass Linux trotz Crash die Daten gefahrlos auf Flash- oder Harddisk speichern kann. Ramoops – wie der Name bereits andeutet – speichert auch Oops-Nachrichten ab, die sich in den Kernel-Logdateien (»/var/log/kern.log«) oder mit Hilfe des »dmesg«-Kommandos studieren lassen.

Ursachenforschung

Wie bei »panic()« sammelt der Kernel beim Oops – so gut es geht – nützliche Informationen. In Listing 4 ist erkennbar, dass eine Codesequenz versucht hat, auf die unerlaubte Null-Adresse zuzugreifen. Für die Ursachenforschung ist auch Zeile 5 wichtig. Die »[#1]« zeigt an, dass dies die erste Oops-Nachricht ist.

Listing 4: Schluckauf-Nachricht vom Kernel (gekürzt)

```
01 oops from the module
02 Unable to handle kernel NULL pointer dereference at virtual address 00000000
03 pgd = c0d3f037
04 [00000000] *pgd=26dd2835, *pte=00000000, *ppte=00000000
05 Internal error: Oops: 817 [#1] SMP ARM
06 Modules linked in: mymodule(O+)
07 CPU: 0 PID: 1544 Comm: insmod Tainted: G           O      4.17.3-v7+ #1
08 Hardware name: BCM2835
09 PC is at my_oops_init+0x24/0x1000 [mymodule]
10 LR is at irq_work_queue+0x14/0x90
11 pc : [<7f005024>]    lr : [<80214988>]    psr: 60000013
12 sp : b96c1d80  ip : 00000007  fp : b96c1d8c
13 r10: b7d12c24  r9 : 0000000c  r8 : 00000001
14 r7 : 00000000  r6 : 80d04d48  r5 : 7f005000  r4 : 7f002000
15 r3 : d43e4d5b  r2 : d43e4d5b  r1 : 00000000  r0 : 00000000
16 Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment user
17 Control: 10c5387d  Table: 3239406a  DAC: 00000055
18 Process insmod (pid: 1544, stack limit = 0xdd41501f)
19 Stack: (0xb96c1d80 to 0xb96c2000)
20 1d80: b96c1e04 b96c1d90 80102fe0 7f00500c 00000000 014000c0 801b3d14 0000000c
22 [...]
23 [<7f005024>] (my_oops_init [mymodule]) from [<80102fe0>] (do_one_initcall+0x50/0x214)
24 [<80102fe0>] (do_one_initcall) from [<801b3d50>] (do_init_module+0x74/0x224)
25 [<801b3d50>] (do_init_module) from [<801b2d88>] (load_module+0x1f34/0x25ec)
26 [<801b2d88>] (load_module) from [<801b3694>] (sys_finit_module+0xd8/0xe8)
27 [<801b3694>] (sys_finit_module) from [<80101000>] (ret_fast_syscall+0x0/0x28)
28 Exception stack(0xb96c1fa8 to 0xb96c1ff0)
29 1fa0:                   20c3b400 00000002 00000003 0002cd30 00000000 00000000
30 1fc0: 20c3b400 00000002 0003f040 0000017b 00000000 000297d8 00000002 00000003
31 1fe0: 7eff75f0 7eff75e0 0002212c 76e6c4f0
32 Code: e301004c e3470f00 eb45db0a e3a00000 (e5800000)
33 ---[ end trace 3ea513722a169319 ]---
```

Häufig zieht ein Fehler diverse andere Fehler, sprich noch mehr Oops-Nachrichten, nach sich. Aber da jede erfolgreiche Fehlersuche bei der ersten Meldung beginnt, soll die Untersuchung bei der »[#1]«-Meldung starten. Und tatsächlich: In Zeile 9 ist die Funktion (»my_oops_init()«) als Auslöser für den Bug, implementiert im Kernelmodul »mymodule«, erkennbar. Dem Entwickler gibt dann insbesondere der im Listing ab Zeile 23 beginnende Stack-Trace noch nützliche Informationen. Mehr Hinweise zur Auswertung finden sich bei [4].

Auf Erden beherrschbar

Wegen der hohen Verbreitung von eingebetteten Linuxen begegnet auch Otto Normalverbraucher ab und an einer Kernel-Panic oder einer Oops-Meldung. Ruhmlose Bilder von abstürzenden Mülltonnen oder auch panischen Fahrinformationssystemen sind per Suchmaschine schnell gefunden [5].

Auch irdische Tesla-Fahrer berichten, dass das im Fahrzeug verbaute Linux-System ab und an einfriert und – selten zwar – abstürzt. Alle Fahr- und Bremsfunktionen bleiben dabei aber vollständig erhalten, und ein gleichzeitiges Drücken der Lenkradtasten über mehrere Sekunden sorgt für den Reboot. Bei Panic kein Grund zur Panik also.

Tesla-Pilot Starman jedoch, der seinen Namen einem David-Bowie-Song verdankt, hat allen Grund zur Panik: Die Akkus seines Wagens sind längst leer und er wird sein Ziel, den Mars, wohl nie erreichen – ihm nutzt Pstore nix. *(jk)* ■

Infos

[1] Ftrace: [https://www.kernel.org/doc/Documentation/trace/]
[2] Raspberry Pi Foundation, „Kernel Building“: [https://www.raspberrypi.org/documentation/linux/kernel/building.md]
[3] „Using ramoops/pstore to capture kernel panics“: [https://www.raspberrypi.org/forums/viewtopic.php?t=199047]
[4] Marian Marinov, „Linux Kernel Crash-dump“: [https://www.slideshare.net/azilian/linux-kernel-crashdump]
[5] „Trash Bin Kernel Panic“: [https://www.pixelstech.net/fun/138-Trash-Bin-Kernel-Panic]