Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Hausarbeit

Debugging im Linux Kernel

vorgelegt von

Marc David

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Informatik
Matrikelnummer: 6824297

Betreuer: Anna Fuchs

Hamburg, 03.03.2018

1 Debugging im Linux Kernel

Abstract

Jedem der sich mit IT beschäftigt, ist bekannt, dass jede Software Fehler enthält. Betriebssysteme nutzen Menschen ständig. Wie reagiert der Linux Kernel auf typische Programmierfehler? Dazu wird ein gegebenes Kernel-Modul analysiert.

1.1 Grundlagen

1.1.1 Debugging

"Debugging is the process of finding and resolving defects or problems within the program that prevent correct operation of computer software or a system [21]". Das Ziel beim Programmieren ist es, dass der Computer ein gewünschtes Verhalten zeigt. Er soll eine Aufgabe unter bestimmten Rahmenbedingungen reproduzierbar ausführen. Dies kann natürlich, aufgrund einer Vielzahl von Gründen wie etwa Logikfehler, Programmierfehler, falsche Benutzung von Bibliotheken etc. fehlschlagen. Nun gibt es mehrere Möglichkeiten einen Fehler zu finden. Sehr hilfreich ist es, wenn der Fehler reproduziert werden kann, sodass sich eingrenzen lässt wo sich der Fehler befindet und ob die vorgenommen Änderungen das beobachtbare Verhalten korrigiert haben. Im Userspace gibt es jede Menge Tools, die einem das Debuggen sehr erleichtern können, wie z. B. Print-Statements, Debugger, Logs, Dumps uvm.

1.1.2 Linux Kernel Grundlagen

Ein Betriebssystem hat zwei Hauptaufgaben, nämlich die Abstraktion der Hardware für Programme sowie die Ressourcenverteilung/-verwaltung [1].
Dazu gibt es noch die Unterscheidung zwischen Kernel Space und User Space. Code der außerhalb des Kernels arbeitet, wird als User Space Code bezeichnet. Mit Kernel Space ist Code gemeint, der innerhalb des Kernels arbeitet. Dieser hat mehr Rechte als ein normales Programm. CPUs unterscheiden Privilegien Stufen wie Ring0 Ring1, ..., Ring3 [24]. Unter Linux werden nur zwei Level verwendet, nämlich Ring0 (Userspace) und Ring 3 (Kernel Space). Prozesse im Userspace können z. B. nicht auf den Speicher des Kernels zugreifen.

Damit ein Userspace Programm z. B. eine Datei öffnen kann, werden System Calls verwendet. Dazu wechselt die CPU auf den Ring0 Modus. Zuerst wird geprüft, ob das Programm die benötigte Berechtigung hat. Falls ja, führt die CPU den System Call

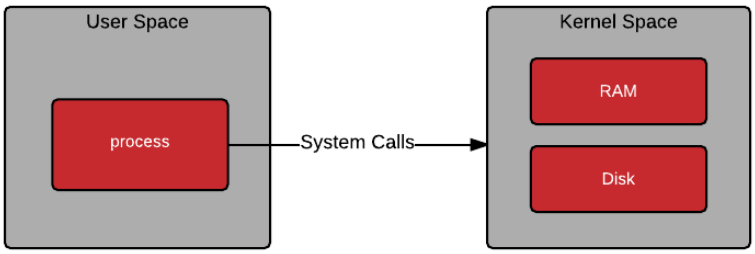

Figure 1.1: Visualisierung Zusammenhang Kernel Space, User Space [25]

aus und das Programm bekommt die Ergebnisse zurück. Jedes Userspace Programm interagiert mit dem Betriebssystem, um beispielsweise Dateien zu öffnen, zu editieren uvm. Um zu zeigen welche Funktionen des Kernels ein normales User Space Programm verwendet, kann *strace* genutzt werden[6].

1.1.3 Kernel Module

Kernel Module bieten die Möglichkeit zur Laufzeit des Betriebssystems Code hinzuzufügen und zu entfernen. Das hat den Vorteil, dass der Linux Kernel z. B. nicht erneut kompiliert werden muss, wenn das Verhalten des Kernels verändert werden soll. Mit ihnen können Aufgaben realisiert werden, die ein normales Programm nicht bietet, wie z. B. einen Gerätetreiber, oder eine Firewall. Auch können System Calls verändert werden, so das z. B. bei einem open System-Call für ein bestimmtes Dateiformat (mp3) immer die gleiche Datei zurückgegeben wird [6]. Ein Kernel Modul wird wie der Linux-Kernel in C geschrieben, wobei viele Bibliotheken nicht verfügbar sind. Außerdem werden spezielle Methoden für den Kernel wie printk oder kmalloc, genutzt.
Im Code eines Kernel-Moduls, befindet sich eine init und exit Funktion. Diese werden mit *"module_init"* und *"module_exit"* entsprechend aufgerufen. Die Methoden werden jeweils einmal ausgeführt, nämlich wenn das Modul in den Kernel einfügt und wieder entfernt wird. Das ist ein weiterer Unterschied zu einem normalen Userspace Programm, welches sequentiell ausgeführt wird. Ein Kernel-Modul registriert sich beim Kernel für ein Event [19].
Um Kernel Modul in den Kernel einfügen muss die C Datei zuerst kompiliert werden. Das Makefile ist anders als bei einem normalen C Programm. Das Ergebnis ist eine *.ko* Datei. Diese wird mit den Befehlen *"insmod"* oder *"modprobe"* in den Kernel eingefügt. Beim Einfügen des Modules wird die init Funktion ausgeführt. Das Kernel Modul kann mit *"rmmod"* wieder aus dem Kernel entfernt werden. Dabei wird die exit Funktion aufgerufen. Alle im Kernel vorhanden Module können mit *"lsmod"* angezeigt werden.

1.1.4 Kernel Oops

Bei einem Kernel Oops ist es zu einem Fehler im Kernel gekommen. Der Kernel-Prozess der für diesen fehlerhaften lokalen Zustand verantwortlich ist, wird beendet, das System läuft weiter. Die Zuverlässigkeit des Systems ist beeinträchtigt[4]. Die Fehlermeldung enthält den Call Trace, Registerinformationen und Module, die der Kernel zu dem Zeitpunkt geladen hatte etc.

1.1.5 Kernel Panic

Eine Kernel Panic ist vergleichbar mit einem Programmabsturz im Userspace Programm, wobei die Folgen weit schlimmer sind. Ein normales Programm wird einfach gekillt. Bei einer Kernel Panic dagegen befindet sich das Betriebssystem in einen inkonsistenten Zustand und ein Neustart wird als einzige Lösung angesehen. Eine Kernel Panic ist vergleichbar mit einem Windows Bluescreen. Um Dateiverluste oder ähnliches zu verhindern sind keine Zugriffe auf Dateisysteme mehr möglich.

1.2 Werkzeuge

Printing

Das einfachste Werkzeug zum Debuggen sind Print Statements. Im normalen C-Userspace Code wird "printf" verwendet, während im Kernel *"printk"* genutzt wird. In einem normalen Programm, werden die Print Nachrichten einfach auf die Konsole ausgegeben, die das Programm gestartet hat.

```
1 printk(KERN_INFO "Hallo Welt");
```

Listing 1.1: Printk

Ein weiter Unterschied sind die Loglevel, welche von "Kernel Emergency" bis "Kernel Debug" reichen können.
Diese Nachrichten werden dann zuerst in einem Systembuffer gespeichert. Sie können z. B. mit "dmesg" ausgelesen werden. Diese Nachrichten werden dann in "/var/log/messages" aufbewahrt. Eine andere Möglichkeit die Print Nachrichten des Kernel zu erhalten ist, den Konsolen Log Level anzupassen, so dass diese einfach im Terminal auftauchen.

```
1 echo 8 > /proc/sys/kernel/printk
```

Listing 1.2: Konsolen Log Level

Eine weitere Möglichkeit auf die Print Statements zuzugreifen ist es, die Datei "/var/log/messages" auszulesen. Die Print Statements werden per Default nicht persistent gespeichert, so das sie nach einem Neustart nicht verfügbar sind. Außerdem werden die Print Statements im Falle eine Kernel Panic nicht auf die Festplatte geschrieben.

KDUMP

KDump ist ein Tool um im Falle einer Kernel Panic Informationen zu sammeln und diese zu speichern. Dies ist nötig, da nachdem eine Kernel Panic aufgetreten ist, keine Zugriffe auf Festplatten etc. möglich sind. Stattdessen wird ein separater Kernel gebootet. Diesem wurde schon beim Systemstart ein separater Speicherbereich zugewiesen. Dies hat den Nachteil, dass der Arbeitsspeicher auch im normalen Betrieb belegt ist und nicht vom Betriebssystem verwendet werden kann. Bei einer Kernel Panic ruft der Hauptkernel den "Dump-Capture" Kernel auf. Nachdem dieser gebootet ist, sammelt und speichert er eine Vielzahl von Informationen. Diese werden per Default in "/var/crash/" gespeichert. Es ist aber auch möglich den Dump z. B. per SSH an eine weitere Maschine zu versenden. Ein Dump besteht aus zwei Dateien. Einmal eine vmcore Datei, die den Dump beinhaltet, sowie eine vmcore-dmesg.txt Datei, welche alle Print Nachrichten des Kernels enthält.

Crash

Um nun den von Kdump erstellten Dump zu lesen, kann Crash verwendet werden [9]. Crash benötigt zwei Dateien als Argument, einmal die Dump Datei, welche von Kdump erstellt wurde, sowie den kompilierten Linux Kernel. Dieser muss in der gleichen Version vorliegen, wie Dump in der Vmcore Datei. Außerdem muss der Kernel mit der "-g" Option von gcc kompiliert worden sein, welche Debugging Symbole enthält.

- *log* (Zeigt alle Print Nachrichten des Kernels an.)
- *bt* (Zeigt alle was zum Zeitpunkt des Crashes ausgeführt wurde.)
- *ps* (Zeigt Prozesse an die zum Crash Zeitpunkt liefen.)

KDB/KGDB

Es gibt für den Linux Kernel auch einen Debugger, den zu verwenden ist aber etwas komplizierter. Man muss den Linux Kernel neu kompilieren und benötigt zwei (virtuelle) Maschinen, die über eine Serielle Schnittstelle verbunden sind [26]. Dies ist nötig, da z. B. während eines Breakpoints keine Tastatureingaben verarbeitet werden könnten.

1.3 Betrachtung der Funktionen des Kernel Moduls

Anmerkung: Der vollständige Quellcode der Funktionen des Kernelmoduls befindet sich im Anhang.

1. Funktion

In dieser Funktion kommt es zu einer Kernel Panic. Im den Crashdump, ist die Fehlermeldung: *"BUG: unable to handle kernel NULL pointer dereference at (null)"* zu sehen. Mit *log* werden die Print Nachrichten des Kernels angezeigt. Innerhalb des Logs findet

sich, ein Hallo Welt Statement. Dies bedeutet, dass der Kernel mit der Ausführung der Init-Funktion begonnen hat. Darauf folgt die obige Fehlermeldung. Auch der Instruction Pointer wird angegeben.

```
1 IP: [<ffffffffa04e0043>] first_error+0x13/0x30
    ↪ [broken_module] [...]
2 Oops: 0002 [#1] SMP
```

Nun kann herausgefunden werden, in welcher Zeile unsere Funktion die Kernel Panic verursacht hat, da der Instruction Pointer bekannt ist. Das Crash-Kommando *bt -l* liefert uns für Kernel Code Zeilennummern. Da unser Fehler sich aber im Kernel Modul ereignet, wird GDB benötigt. Im Print-Log finden ist Symbol und Offset innerhalb der Funktion und Range zu finden[20].

```
1 [<ffffffffa04c6017>] broken_init+0x17/0x1b [broken_module]
```

Dazu wird GDB mit dem kompilierten Kernel Modul geöffnet. Dann wird das folgende Kommando eingegeben: *"list *(first_error+0X13)"*
Als Antwort bekommen wird den Quellcode und die Zeile 14 (8) genannt. In dieser Zeile wird die Zuweisung *a = b vorgenommen. Der Pointer a zeigt also auf 0. Der Fehlercode 0002 -> 0b0010 bedeutet: Not instruction fetch | Kernel mode | Write | Invalid access. [18]

2. Funktion

Diese Funktion hat in ihrem ursprünglichen Zustand bei meinen virtuellen Maschinen zu keinem Fehler geführt. Deshalb wurde der Maximalwert der Zählvariablen auf 100 zu erhöhen. Es wurden sowohl eine Kernel Panic, als auch mehrere Kernel Oops beobachtet. Manchmal ist die Init-Funktion des Kernel Moduls ohne Fehler ausgeführt worden.
Bei der Panic lautetet die von Crash ausgegebene Fehlermeldung: *"BUG: unable to handle kernel paging request at ffff8804b8bed06c"*. Das Kommando das zu dem Zeitpunkt ausgeführt wurde lautet "udevd" Dies ist ein Daemon des Kernels [23]. Der Fehlercode *Oops: 0000 [#1] SMP gibt an: Not instruction fetch | Kernel mode | Read or Execute | Invalid access.*
Bei einem anderen Hinzufügen des Kernel Moduls kommt es zu folgender Fehlermeldung: *"BUG: Bad page map in process insmod pte:8000000000000000 pmd:bc9e9067"* Beide Fehlermeldungen deuten darauf hin, das Daten des Kernels verändert wurden. Bei Betrachtung des Codes wird schnell klar, dass diese Funktion einen Buffer Overflow verusacht. Sie schreibt über die Grenzen des allozierten Speicherbereichs hinaus. Die Ergebnisse dessen sind natürlich davon abhängig, was überschrieben wird.

```
1 int second_error(void) {
2     //[..]
3     int *b = kmalloc(sizeof(char) * 16, GFP_KERNEL);
4     for (a = 0; a < 16; a++) {
5         b[a * 16] = 0;
```

```
6       //[..]
```

3. Funktion

Diese Funktion hat auf meinen virtuellen Maschinen zu keinem Fehler geführt. Es wurden sowohl count als auch msleep erhöht, wobei dies zu keinem Fehler geführt hat.
Im Quellcode ist zu sehen, dass die Funktion sich Speicher alloziert, diesen dann anschließend freigibt, wartet und wieder in den ehemals allozierten Speicherbereich hineinschreibt. Dies führt genau dann zu einem Fehler, wenn der Speicher innerhalb der Wait-Time von einem anderen Betriebssystemprozess verwendet wird. Ein Grund könnte sein, dass die VMs wegen KDump mit 3GB Ram ausgestattet worden sind. Aber auch eine virtuelle Maschine mit 256 MB RAM hat zu keinen Fehlermeldungen in */var/log/messages* geführt. Vermutlich hat während der Wait Time kein anderer Prozess im Kernel den beschriebenen Speicher zugewiesen bekommen und verwendet. Das Ergebnis wäre wahrscheinlich vergleichbar mit dem Fehler in Funktion 2.

```
1  int third_error(void) {
2      long *a = kmalloc(sizeof(*a) * count, GFP_KERNEL);
3      //[...]
4      kfree(a);
5      msleep(1000);
6
7      for (i = 0; i < count; i++) {
8      a[i] = 0;
9      //[...]
```

4. Funktion

Diese Funktion führt zu einer Kernel Panic. Crash liefert uns folgende Informationen:

```
1  kernel BUG at mm/slab.c:524!
2  invalid opcode: 0000 [#1] SMP
```

Nun scheint der Kernel zu glauben, dass ein Fehler im Kernel existiert. Natürlich ist es möglich sich den Quellcode des Kernels anzuschauen.
Der Prozessor meldet, dass er einen ungültigen Befehl erhalten hat. Der Fehlercode bedeutet folgendes: ““*Not instruction fetch | Kernel mode | Read Or Execute | Invalid access*”” [18]. Es gab also einen ungültigen Zugriff, lesend oder schreibend im Kernel Modus. Im Kernel Log lässt sich folgende Nachricht finden:

```
1  [<ffffffffa04d00aa>] fourth\_error+0x4a/0x60
     ↪ [broken\_module]
```

Wird Symbol und Offset in GDB angeben, ist das zweite *kfree* das Ergebnis.

```
1 void fourth_error(void) {
2     [...]
3     kfree(message);
4     kfree(message);
5 }
```

Listing 1.3: Kernel-Modul Funktion 4

5. Funktion

Bei Ausführung dieser Methode auf einer VM mit 256 MB RAM kam es zu einer Kernel Oops. Im Log ist dass Hallo Welt zu finden. In der folgenden Zeile ist folgende Fehlermeldung zu sehen:

```
1 insmod: page allocation failure. order:10, mode:0xd0
2 Pid: 2762, comm: insmod Not tainted
    ↪ 2.6.32-696.13.2.el6.x86_64 #1
```

Die Speicherverwaltung in Betriebssystemen funktioniert mithilfe des Konzeptes von Virtual Memory. Der Speicher wird in sogenannte Pages aufgeteilt. Möchte ein Programm sich mit *kmalloc/malloc* Speicher allozieren, bekommt es die entsprechende Anzahl an Pages. Die Pages können auch auf die Festplatte ausgelagert werden. Der Kernel reserviert Memory Pages nur in Zweipotenzen. Die 10 bei "order 10" ist der Exponent. Der Kernel wollte also 1024 Seiten von jeweils 4096 Bytes allozieren. Mit *"getconf PAGESIZE"* wird die Seitengröße des Kernels in Bytes angezeigt. Der Modus *0xc0d0* gibt an, dass innerhalb des Kernels kalloc aufgerufen worden ist. Im Call Trace wird die Funktion fifth erwähnt. So kann mit Hilfe von GDB in Erfahrung gebracht werden in welcher Zeile der Fehler aufgetreten ist. Dazu kann *"gdb list *(fifth_error+0x1f)"* verwendet werden. Das Ergebnis ist c:73 (Eine Zeile nach dem kmalloc). Danach folgen noch einige weitere Informationen zum Speicher.

```
1 void fifth_error(void) {
2     long long *a = kmalloc(sizeof(*a) * 128 * 1024,
        ↪ GFP_KERNEL);
3 }
```

Listing 1.4: Kernel-Modul Funktion 5

1.4 Zusammenfassung

Es lässt sich feststellen, dass Debugging im Kernel von Linux keine einfache Angelegenheit ist. Fehler können sehr einfach entstehen, indem beispielsweise in den Speicher von anderen Kernel-Anwendungen hineingeschrieben wird. Dies ist bei Userspace Programmen nicht möglich. Trotzdem gibt es einige Möglichkeiten Fehler genauer zu analysieren, wie Kdump, Crash, Printing etc.

1.5 Quellen

1. Andrew S. Tanenbaum: Modern Operating Systems (3rd Edition)
2. Daniel P. Bovet, Marco Cesati-Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
3. Linux Device Drivers, (3rd Edition), Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman (Chapters 2, 4 8) http://lwn.net/Kernel/LDD3/
4. Is Linux Kernel Oops Useful or Not? Takeshi Yoshimura, Hiroshi Yamada,Kenji Kono https://www.usenix.org/system/files/conference/hotdep12/hotdep12-final19.pdf
5. Linux kernel debugging for sysadmins, Minto Joseph FrOSCon 2017, Eingesehen am 8.12.17: http://media.ccc.de/v/froscon2017-1925-linux_kernel_debugging_for_sysadmins#video
6. „You can be a kernel hacker!", Eingesehen am 8.12.17: http://jvns.ca/blog/2014/09/18/you-can-be-a-kernel-hacker/
7. The Linux Kernel Module Programming Guide , Eingesehen am 8.12.17: http://www.tldp.org/LDP/lkmpg/2.6/html/x427.html
8. Redhat Enterprise Linux Deployment Guide: Chapter 32, Eingesehen am 8.12.17: http://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/deployment_guide/ch-kdump
9. White Paper: Red Hat Crash Utility, Eingesehen am 8.12.17: http://people.redhat.com/~anderson/crash_whitepaper/
10. Documentation for Kdump - The kexec-based Crash Dumping Solution, Eingesehen am 8.12.17: http://www.kernel.org/doc/Documentation/kdump/kdump.txt
11. Decoding the Linux kernel's page allocation failure messages, Eingesehen am 8.12.17: http://utcc.utoronto.ca/~cks/space/blog/linux/DecodingPageAllocFailures
12. Using kgdb / gdb, Eingesehen am 8.12.17:http://www.kernel.org/doc/html/v4.13/dev-tools/kgdb.html#using-kgdb-gdb
13. Using Modprobe, Eingesehen am 8.12.17: http://www.linuxforums.org/forum/applications/194343-modprobe-fatal-module-hello-ko-not-found.html
14. Architecting Containers Part 1:Why Understanding User Space vs. Kernel Space Matters, Eingesehen am 8.12.17: http://rhelblog.redhat.com/2015/07/29/architecting-containers\-part-1-user-space-vs-kernel-space/
15. Options for Debugging Your Program, Eingesehen am 8.12.17: http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html Professional Linux kernel architecture Wolfgang Mauerer (2008)
16. Kernel Parameter für Linux, Eingesehen am 8.12.17: http://www.kernel.org/doc/Documentation/admin-guide/kernel-parameters.txt

17. Analyzing Linux kernel crash dumps with crash, Eingesehen am 8.12.17: http://www.dedoimedo.com/computers/crash-analyze.html

18. Linux Kernel: memory corruption - debug tricks, Eingesehen am 8.12.17: http://helenfornazier.blogspot.de/2015/07/linux-kernel-memory-corruption-debug.html

19. Writing a Linux Kernel Module — Part 1: Introduction, Eingesehen am 26.2.18: http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/

20. Understanding a Kernel Oops! Surya Prabhakar, Eingesehen am 27.2.18: http://opensourceforu.com/2011/01/understanding-a-kernel-oops/

21. Wikpedia Definition Debugging, Eingesehen am 27.2.18: https://en.wikipedia.org/wiki/Debugging

22. addr2line on kernel module, Eingesehen am 28.12.18: http://stackoverflow.com/questions/6151538/addr2line-on-kernel-module

23. udevd Eingesehen am 28.12.18: http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev/udevd.html

24. Wolfgang Mauerer: Professional Linux Kernel Architecture

25. Bild User Space vs Kernel Space, Eingesehen am 28.2.18 https://rhelblog.files.wordpress.com/2015/07/user-space-vs-kernel-space-simple-user-space.png

26. Using kgdb, kdb and the kernel debugger internals, Eingesehen am 2.3.18 https://www.kernel.org/doc/html/v4.13/dev-tools/kgdb.html

1.6 Quellcode der Funktionen des Kernel Moduls

```
1 int first_error(void) {
2     int *a = (int *) 0;
3     int b = 5;
4     msleep(500);
5     *a = b;
6     return *a;
7 }
```

Listing 1.5: Funktion 1

```
1 int second_error(void) {
2     int a;
3     int *b = kmalloc(sizeof(char) * 16, GFP_KERNEL);
4     for (a = 0; a < 16; a++) {
5         b[a * 16] = 0;
6     }
7     kfree(b);
8     return a;
```

```
}
```

Listing 1.6: Funktion 2

```
void third_error(void) {
    int i;
    int count = 1000;
    long *a = kmalloc(sizeof(*a) * count, GFP_KERNEL);
    for (i = 0; i < count; i++) {
        a[i] = i;
    }
    kfree(a);
    msleep(1000);

    for (i = 0; i < count; i++) {
        a[i] = 0;
    }
}
```

Listing 1.7: Funktion 3

```
void fourth_error(void) {
    char *message = kmalloc(sizeof(*message) * 6, GFP_KERNEL);
    message = "hello";
    printk(KERN_ERR
    "%s\n", message);
    msleep(500);

    kfree(message);
    kfree(message);
}
```

Listing 1.8: Funktion 4

```
void fifth_error(void) {
    long long *a = kmalloc(sizeof(*a) * 128 * 1024, GFP_KERNEL);
}
```

Listing 1.9: Funktion 5