

Malware-Analyse und Reverse Engineering

10: Systemaufrufe und Tracing

1.6.2017

Prof. Dr. Michael Engel

Überblick

Themen:

- Systemaufrufe in Linux
- Verfolgen von Systemaufrufen: strace/ptrace
- Verhaltensanalyse
- Maschinelles Lernen und Virenentdeckung

Systemaufrufe in Linux

Bisher: Funktionsaufrufe

- Aufruf von Code in dem Adressraum, in dem auch der Aufrufer ausgeführt wird
- Parameterübergabe in Registern (x64)/auf dem Stack (x86)
- Danach Ausführen einer CALL-Instruktion
- Rückkehr aus Funktion mit RET-Instruktion

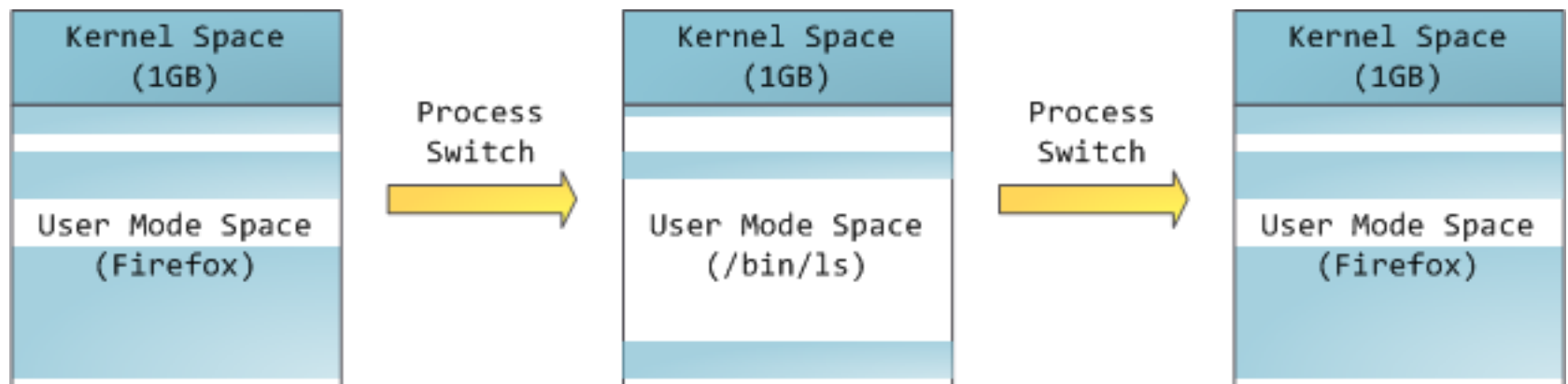
Systemaufruf: Ausführen einer Funktion des Betriebssystemkerns

- Funktionen, die Zugriff auf die Hardware erfordern
 - z.B. Lesen von der Tastatur
- Funktionen, die gemeinsame Ressourcen betreffen
 - z.B. Zugriff auf Dateien in einem Dateisystem

Wiederholung: Adressraumaufteilung bei x86

Kernel (bei 32 Bit x86) belegt 1GB am Ende des Adressraums

- Dort liegen Funktionen und Daten des Kernels
- Reicht nicht einfach ein CALL in diesen Adressraum, um Kernelfunktionen aufzurufen?
 - normalerweise nicht (selten etwas ähnliches \Rightarrow vdso)

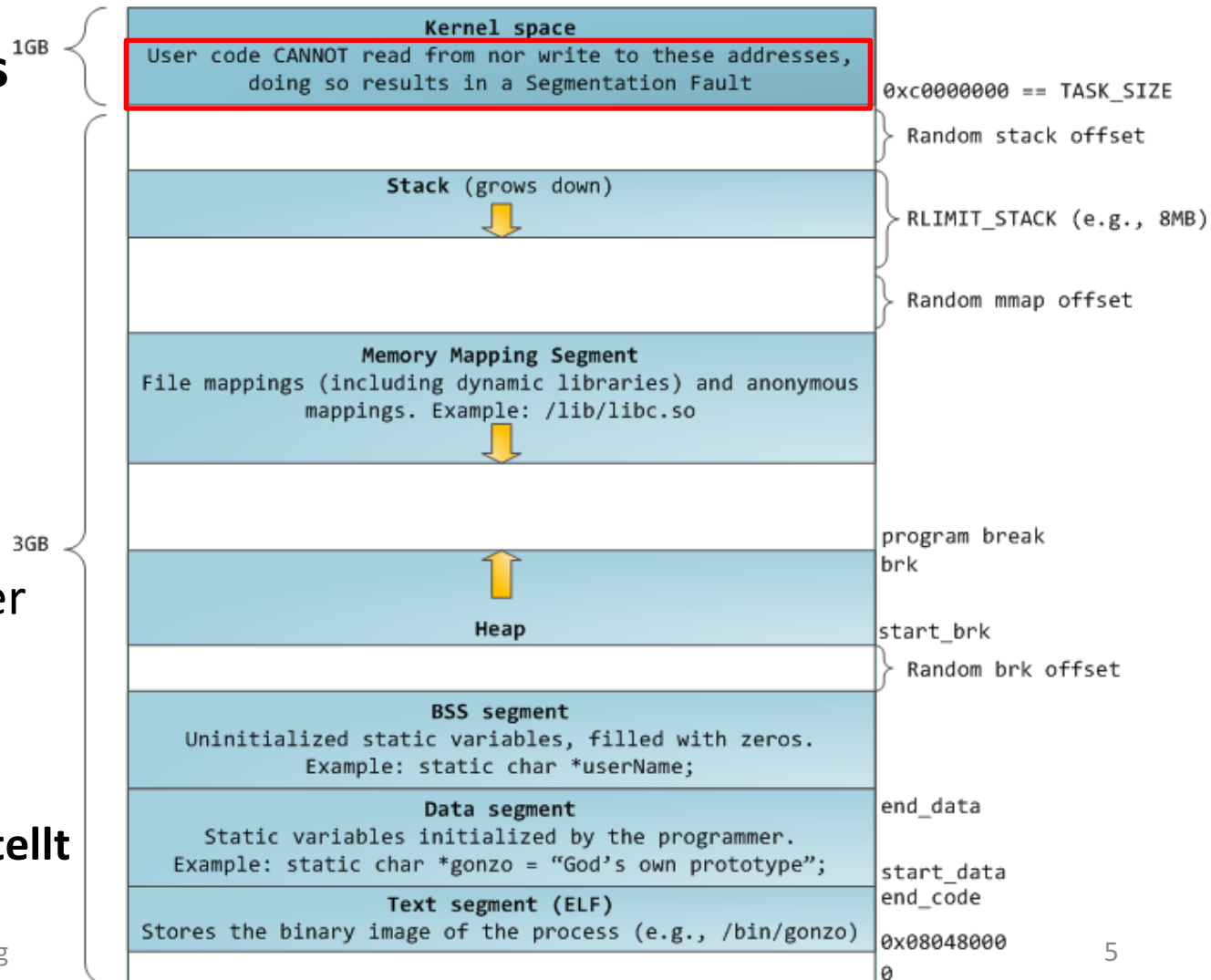


Schutz von Kernel-Code und -Daten

Zugriff auf oberes 1 GB Adressraum ist beschränkt

- Lesen (also auch Ausführen von Code) und Schreiben wird durch passende Konfiguration der Seitentabellen verhindert

Achtung: 32 bit x86 Linux-System dargestellt



Privilegienebenen und Privilegientransfer

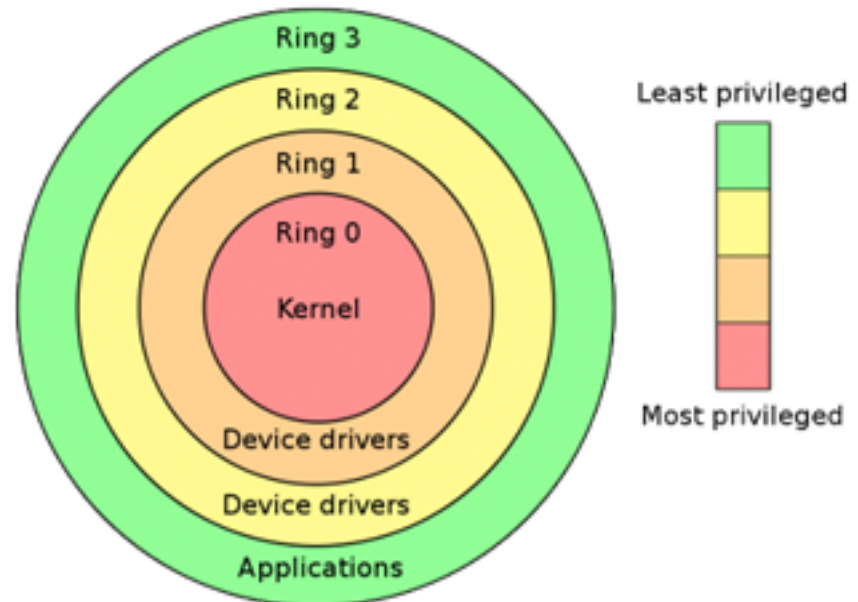
Direkter Zugriff auf Kernel-eigene Daten und Code von User-Mode-Programmen aus nicht möglich

- „User Mode“ \Rightarrow Beschreibung eines Systemzustands
 - Jeder ausführbare Code wird im Kontext eines Modus ausgeführt
 - Davon abgeleitet: Zugriffsrechte auf Ressourcen
 - z.B. Speicherseiten, I/O-Adressen, privilegierte Maschinenbefehle (wie z.B. Interruptsperrern)
 - Anderer Zustand: „Kernel Mode“

Privilegienebenen bei x86/x64

Prozessor implementiert sog. „Schutzringe“

- Niedrigere Ringnummer = mehr Privilegien
- **0 = Kernel** = voller Zugriff *
- 1 und 2 = reduzierter Zugriff
 - Linux nutzt diese nicht
 - Windows: Grafiktreiber
- **3 = Benutzerprogramme**
 - Dazu zählen auch Systemverwaltungstools für den Administrator, z.B. mkfs!



* Wenn eine CPU Hardware-Virtualisierung unterstützt, existiert noch ein Ring -1 für den Hypervisor, z.B. Xen

Systemaufrufe: Überblick über die Funktionalität

Verschiedene Gruppen von Funktionalitäten

- Dateiverwaltung: open, read, write, seek, close, ...
- Prozessverwaltung: fork, exec, kill, exit, ...
- Rechteverwaltung: setuid, setgid, getgid, ...
- Synchronisation/Kommunikation zwischen Prozessen: signal, wait, pause, ipc, ...
- Systemverwaltung: swapon, reboot, syslog, ...
- und viele weitere:
 - Insgesamt über 300 Systemaufrufe (Kernel 4.11)
 - In Systemaufruftabelle: Eindeutige Nummer und Name sowie Funktionszeiger für jeden Systemaufruf

https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl

Systemaufruftabelle

```
1 #
2 # 64-bit system call numbers and entry vectors
3 #
4 # The format is:
5 # <number> <abi> <name> <entry point>
6 #
7 # The abi is "common", "64" or "x32" for this file.
8 #
9 0      common  read      sys_read
10 1      common  write     sys_write
11 2      common  open      sys_open
12 3      common  close     sys_close
13 4      common  stat      sys_newstat
14 5      common  fstat     sys_newfstat
15 6      common  lstat     sys_newlstat
16 7      common  poll      sys_poll
17 8      common  lseek     sys_lseek
```

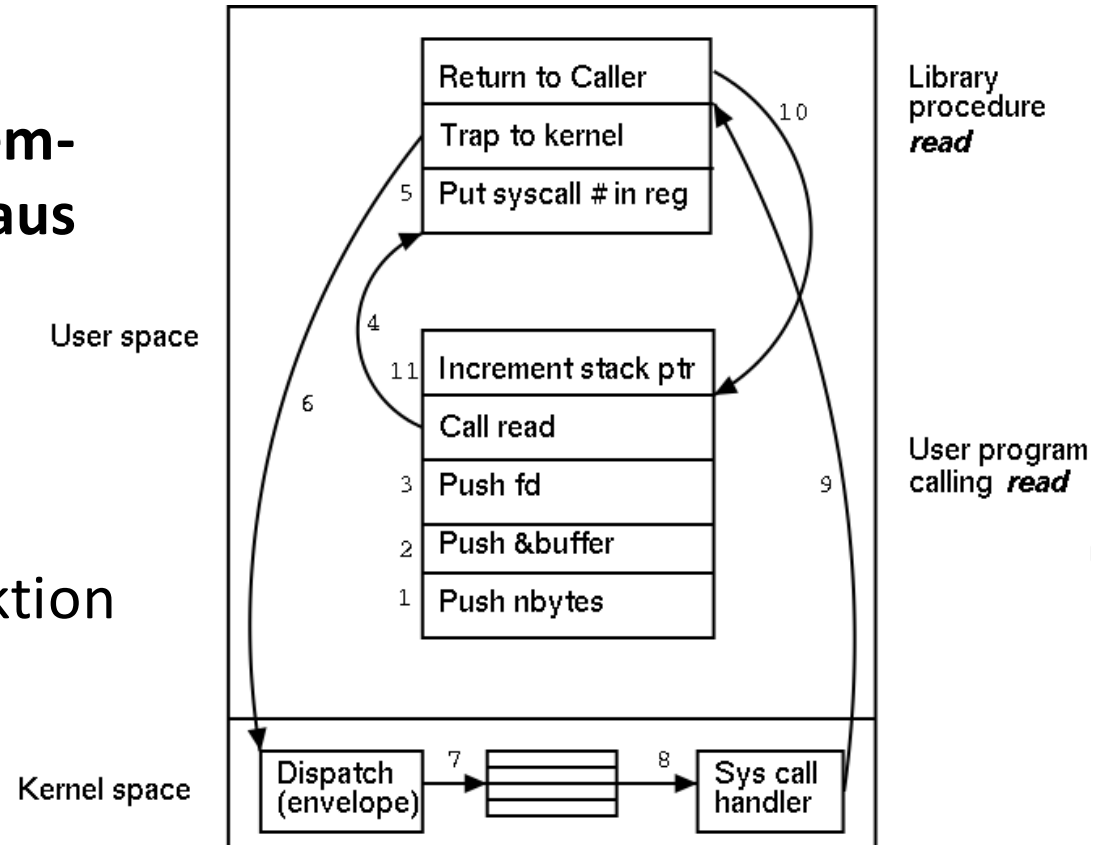
Beispiel für Systemaufruf: read

READ(2)	Linux Programmer's Manual	READ(2)
NAME	top	
	read - read from a file descriptor	
SYNOPSIS	top	
	<pre>#include <unistd.h> ssize_t read(int fd, void *buf, size_t count);</pre>	
DESCRIPTION	top	
	<p><code>read()</code> attempts to read up to <code>count</code> bytes from file descriptor <code>fd</code> into the buffer starting at <code>buf</code>.</p> <p>On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and <code>read()</code> returns zero.</p> <p>If <code>count</code> is zero, <code>read()</code> <i>may</i> detect the errors described below. In the absence of any errors, or if <code>read()</code> does not check for errors, a <code>read()</code> with a <code>count</code> of 0 returns zero and has no other effects.</p> <p>According to POSIX.1, if <code>count</code> is greater than <code>SSIZE_MAX</code>, the result is implementation-defined; see NOTES for the upper limit on Linux.</p>	

Systemaufrufe und die libc

C-Programme führen Systemaufrufe meist nicht direkt aus

- „Umweg“ über die libc: sog. **system call stubs**
- Aufruf von `read(...)` in C-Programm springt in entsprechende libc-Funktion
- Diese führt dann den Systemaufruf aus
- Warum der Umweg?
 - Extrafunktionalität (z.B. bei `exit/atexit`)
 - Kompatibilität: mehrere Methoden für Systemaufrufe



Privilegienübergang bei Systemaufrufen

Wie kann nun ein Systemaufruf durchgeführt werden?

- Adresse der Funktion in Systemaufruftabelle
 - Diese ist aber nur intern im Kernel verfügbar
- Wir wissen: Code im User mode kann Funktionen nicht direkt per Call aufrufen
 - *Die libc wird aber auch im User mode ausgeführt!*
 - Als shared library im Adressraum des jew. Prozesses

Lösung: spezielle Maschineninstruktion

- Maschinenbefehl **SYSCALL**
 - Privilegienwechsel in Ring 0
 - Sprung an definierte Adresse („Dispatcher“)
 - Von Kernel in CPU-Spezialregister IA32_LSTAR abgelegt

SYSCALL in Assembler (1)

Alle Aufrufe von SYSCALL „landen“ an einer Adresse im Kernel

- Kernel muss gewünschte Funktionalität kennen:
 - **Systemaufrufnummer** und **Parameter** des Systemaufrufs
 - Aufrufender Prozess muss diese jeweils in Prozessorregistern übergeben
 - rax: Nummer des Systemaufrufs
 - rdi, rsi, rdx, rcx, r8, r9: Parameter 1...6
 - Rückgabewert wird in rax zurückgegeben
- Kernel liest nach Ausführen der SYSCALL-Instruktion Wert in rax
 - Nutzt diesen als Index in Systemaufruftabelle
 - ...und springt zu dort hinterlegtem Funktionszeiger

SYSCALL in Assembler (2)

Beispiel: write() – Schreiben von Bytes in Datei

Prototyp:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- Parameter:
 - **rax**: Nummer des Systemaufrufs = 1
 - **rdi**: fd – **rsi**: buf – **rdx**: count – **rcx, r8, r9**: unbenutzt

```
_start:                # Programm beginnt immer bei _start
    mov     $1, %rax    # Systemaufruf 1 = write
    mov     $1, %rdi    # Dateihandle 1 ist "stdout"
    mov     $message, %rsi # Adresse von auszugebendem String
    mov     $13, %rdx   # Anzahl auszugebender Bytes (13)
    syscall            # Systemaufruf
    ...
    .data              # String ins Datensegment
message: .ascii "Hello, world\n"
```

Schreiben von Assemblerprogrammen

Achtung:

- Bei x86/x64-Prozessoren existieren **zwei** unterschiedliche Konventionen für die Schreibweise von Assemblerbefehlen
 - Intel-Standard: Verwendet von Assembler **nasm**
 - AT&T-Standard: Verwendet von GNU-Tools (**gas, objdump**)
- Wir verwenden den AT&T-Standard
 - Intel verwendet andere Konventionen für Registernamen und **umgekehrte Reihenfolge von Quelle und Ziel**
 - **Also: Aufpassen bei Beispielen aus den Netz!**

Übersetzen von Assemblerprogrammen

Übersetzen mit gcc als front end:

- Erlaubt Verwendung von C(++)-Kommentaren und Makros
- gcc ruft dann in Folge den GNU-Assembler gas auf
- Assemblerprogramm haben (Konvention) Dateiendung „.S“
 - *Achtung: Großbuchstabe S*

```
$ gcc -c meinprog.S           # erzeugt meinprog.o
$ ld -o meinprog meinprog.o  # linkt meinprog.o zu meinprog
$ ./meinprog                  # meinprog ausführen
```

- Diese Methode des Übersetzens vermeidet das Dazulinken nicht benötigter Komponenten wie libc und crt0

Disassembliertes Programm

Im Gegensatz zu C: Assembler fügt keinen Extra-Code ein

- z.B. automatischer Rücksprung aus Funktion mit Return


```
$ objdump -d meinprog
meinprog:      file format elf64-x86-64
```

```
Disassembly of section .text:
```

```
00000000004000b0 <_start>:
```

```
4000b0:      48 c7 c0 01 00 00 00      mov     $0x1,%rax
4000b7:      48 c7 c7 01 00 00 00      mov     $0x1,%rdi
4000be:      48 c7 c6 da 00 60 00      mov     $0x6000da,%rsi
4000c5:      48 c7 c2 0d 00 00 00      mov     $0xd,%rdx
4000cc:      0f 05                    syscall
```

Adresse unseres Strings
im Datensegment, von
Linker eingefügt



Programm „sauber“ beenden (1)

Beim Ausführen...

- Nach SYSCALL-Instruktion stehen zufällige Daten im Textsegment
- Absturz...

```
$ ./meinprog
Hello, world
Segmentation fault (core dumped)
```

- Lösung: Return-Instruktion einfügen? => **stürzt auch ab!**

```
_start:                # Programm beginnt immer bei _start
    mov     $1, %rax    # Systemaufruf 1 = write
    mov     $1, %rdi    # Dateihandle 1 ist "stdout"
    mov     $message, %rsi # Adresse von auszugebendem String
    mov     $13, %rdx   # Anzahl auszugebender Bytes (13)
    syscall            # Systemaufruf
    ret                # Kein Stackkontext vorhanden!!!
    .data              # String ins Datensegment
message: .ascii "Hello, world\n"
```

Programm „sauber“ beenden (2)

Funktionierende Lösung:

- Systemaufruf „exit“ (Nr. 60) verwenden
 - Parameter: Rückgabewert

```
_start:                # Programm beginnt immer bei _start
    mov     $1, %rax    # Systemaufruf 1 = write
    mov     $1, %rdi    # Dateihandle 1 ist "stdout"
    mov     $message, %rsi # Adresse von auszugebendem String
    mov     $13, %rdx   # Anzahl auszugebender Bytes (13)
    syscall            # Systemaufruf
    mov     $60, %rax   # Systemaufruf 60 = exit
    xor     %rdi, %rdi  # Parameter (Rückgabewert) = 0
    syscall            # Systemaufruf exit kehrt nicht zurück!

    .data              # String ins Datensegment
message: .ascii "Hello, world\n"
```

Programm „sauber“ beenden (3)

Funktionierende Lösung:

- Systemaufruf „exit“ (Nr. 60) verwenden
 - Parameter: Rückgabewert
 - In Shell mit „echo \$?“ abfragbar
 - Wert von 0...255

```
$ gcc -c meinprog.S           # erzeugt meinprog.o
$ ld -o meinprog meinprog.o  # linkt meinprog.o zu meinprog
$ ./meinprog                 # meinprog ausführen
Hello, world
$ echo $?                    # Rückgabewert abfragen
0
$
```

Verfolgen von Systemaufrufen: strace (1)

Welche Systemaufrufe und Parameter verwendet ein Programm?

- Werkzeug zum Debuggen/Analysieren: **strace**
- Hier für unser Assembler-Beispielprogramm:

Ausführen
von meinprog

Syscall
"write"

Syscall
"exit"

```
$ strace ./meinprog  
execve("./meinprog", ["./meinprog"], [/* 44 vars */]) = 0  
write(1, "Hello, world\n", 13Hello, world  
) = 13  
exit(0) = ?  
+++ exited with 0 +++
```

argv

Environment-
Variablen

Rückgabewert
von write:
Anzahl geschrie-
bener Zeichen

Dies ist die
Ausgabe unseres
Programms,
diese erscheint
"mittendrin"

Verfolgen von Systemaufrufen: strace (2)

Beispiel: „ls“ – 96 Zeilen Ausgabe – shared libs & Speicherverwaltg:

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 44 vars */]) = 0
brk(NULL)                                = 0x13ab000
access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f6f578c1000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=180082, ...}) = 0
mmap(NULL, 180082, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f6f57895000
close(3)                                  = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260Z\0\0\0\0\0"... , 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=130224, ...}) = 0
mmap(NULL, 2234080, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f6f5747c000
...
```

Verfolgen von Systemaufrufen: strace (3)

Beispiel: „ls“ – 96 Zeilen Ausgabe...

```
$ strace ls
...(einige Zeilen ausgelassen)...

open(".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3 # Aktuelles Dir. öffnen
fstat(3, {st_mode=S_IFDIR|0750, st_size=6, ...}) = 0 # Zugriffsrechte prüfen
getdents(3, /* 6 entries */, 32768) = 160 # Dir-Einträge lesen
getdents(3, /* 0 entries */, 32768) = 0
close(3) = 0 # Directory schließen
fstat(1, {st_mode=S_IFIFO|0600, st_size=0, ...}) = 0 # Ausgabeparam. prüfen
write(1, "core\nsc1\nsc1.S\nsc1.o\n", 21) = 21 # Dateiliste ausgeben
close(1) = 0 # Dateien schließen
close(2) = 0 # (stdout=1, stderr=2)
exit_group(0) = ? # ...und raus hier!
+++ exited with 0 +++
```

Verfolgen von Systemaufrufen: ptrace

strace-Funktionalität auch in eigenen Programmen nutzbar

NAME

ptrace – process trace

SYNOPSIS

```
#include <sys/ptrace.h>
```

```
long ptrace(enum __ptrace_request request, pid_t pid,  
            void *addr, void *data);
```

DESCRIPTION

The `ptrace()` system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

A tracee first needs to be attached to the tracer. Attachment and subsequent commands are per thread: in a multithreaded process, every thread can be individually attached to a (potentially different) tracer, or left not attached and thus not debugged. Therefore, "tracee" always means "(one) thread", never "a (possibly multithreaded) process". Ptrace commands are always sent to a specific tracee using a call of the form

```
ptrace(PTRACE_foo, pid, ...)
```

where `pid` is the thread ID of the corresponding Linux thread.

ptrace-Aufruf (1)

Komplexes API für Prozesstracing und -analyse

```
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

1. Parameter: Auszuführende Funktion:
 - **PTRACE_TRACEME**: aktuellen Prozess tracen
 - **PTRACE_PEEKTEXT, PTRACE_PEEKDATA**:
Wort an Adresse “addr” lesen -> Rückgabewert der Funktion
 - **PTRACE_PEEKUSER**: Wort an Offset “addr” in der USER-Area des Prozesses lesen. Diese enthält u.a. Werte der Prozessorregister

ptrace-Aufruf (2)

1. Parameter: Auszuführende Funktion:
 - **PTRACE_POKETEXT, PTRACE_POKEDATA:**
Wort “data” an Adresse “addr” schreiben
 - **PTRACE_POKEUSER:** Wort “data” an Offset “addr” in der USER-Area des Prozesses lesen (z.B. zum Ändern der Prozessorregister)
 - **PTRACE_GETREGS, PTRACE_GETFPREGS, PTRACE_GETREGSET:**
Register des Prozesses lesen (werden an Adresse “data” kopiert)
– Integer/Floating Point/alle
 - **PTRACE_SETREGS, PTRACE_SETFPREGS, PTRACE_SETREGSET:**
Register des Prozesses schreiben (von Adresse “data”)
 - ...und viele andere mehr, siehe manpage

ptrace-Beispiel (1)

(Auszug, komplett im moodle)

```
int main()
{
    pid_t child;

    child = fork();           // Zu tracenden Kindprozess erzeugen

    if (child == 0) {       // ==0? in Kindprozess
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
                            // diesen tracen
        execl("/bin/ls", "ls", NULL);
                            // das eigentlich interessante Programm
    }
}
...
```

ptrace-Beispiel (2)

```
else {          // sonst: Elternprozess, der Kind tracen will
    long orig_rax, rax;
    long params[3];
    int status;

    while(1) {
        wait(&status); // warte auf Signal von Kind
        if(WIFEXITED(status))
            break;     // wenn Kind beendet => auch beenden
        // RAX == Nummer des Syscalls, mit Peekuser lesen
        orig_rax = ptrace(PTRACE_PEEKUSER, child, 8 * ORIG_RAX, NULL);
        ...
    }
}
```

ptrace-Beispiel (3)

```
...
// RAX == Nummer des Syscalls, mit Peekuser lesen
orig_rax = ptrace(PTRACE_PEEKUSER, child, 8*ORIG_RAX, NULL);
// Wenn Syscall == write
if (orig_rax == SYS_write) {
    // Parameter 0,1,2 (rdi, rsi, rdx) auslesen
    params[0] = ptrace(PTRACE_PEEKUSER, child, 8*RDI, NULL);
    params[1] = ptrace(PTRACE_PEEKUSER, child, 8*RSI, NULL);
    params[2] = ptrace(PTRACE_PEEKUSER, child, 8*RDX, NULL);
    printf("Write called with %ld, %lx, %ld\n",
           params[0], params[1], params[2]);
}
// Weiter Syscalls des Kindprozesses tracen
ptrace(PTRACE_SYSCALL, child, NULL, NULL);
} // Ende von while
} // Ende von else
} // Ende von main
```

Fazit

Systemaufrufe

- Schnittstelle zwischen User und Kernel mode
- Standardmethode zur Verwendung von Kernelfunktionen
- Üblicherweise von libc aufgerufen
- In Assembler: SYSCALL-Instruktion
- Tracing von Systemaufrufen mit strace und ptrace
 - Dies ist die Basis für eine „Überwachung“ von Prozessen
 - ...also z.B. Verhaltensanalyse von Viren
(nächste Vorlesung)

Referenzen

1. Durchsuchbare Tabelle von Linux-Systemaufrufen
 - <https://filippo.io/linux-syscall-table/>
2. S. Forrest, S. A. Hofmeyr, A. Somayaji and T. A. Longstaff, “*A sense of self for Unix processes*”, Proceedings 1996 IEEE Symposium on Security and Privacy, Oakland, CA, 1996, pp. 120-128
 - Erste Ideen zur Verhaltensanalyse von Unix-Prozessen
3. G. Jacob, H. Debar, E. Filiol, “*Behavioral detection of malware: From a survey towards an established taxonomy*”, Journal in Computer Virology 4(3):251-266, August 2008
 - Überblick über >50 Verhaltensanalysen aus Forschung und Industrie