OS-UE

Praktische Übung 2

Michael Pucher Gabriel Gegenhuber

1 Einleitung

Im Gegensatz zur ersten praktischen Übung handelt es sich bei dieser Übung um eine Programmieraufgabe. Im Rahmen dieser Aufgabe wird über verschiedene Schnittstellen mit dem Linux Kernel direkt kommuniziert in programmatischer Form, um die unterliegenden Konzepte von einigen Commandline-Tools der letzten Übung näher zu bringen. Das setzt zwingend voraus, dass man die Programme auf einem Linux-System testet.

Anders als in der letzten Aufgabe legen wir allerdings keine spezifischen Anforderungen an das Linux-System fest. Es steht dir damit frei, eine eigene Distribution, oder z.B. das Windows Subsystem for Linux zu benutzen. Wir werden die Programme auf einem System testen, das ähnlich zur letzten Übung sein wird (sprich *Debian 11.3*). Das bedeutet, man kann auch einfach die VM der letzten Übung für diese Übung weiterverwenden. Da das Schreiben von Code auf der VM selbst eher mühsam ist, können z.B. Visual Studio Code, oder andere Editoren/IDEs verwendet werden, die direkt SSH Zugriff auf ein remote System erlauben.

Für Fragen gilt dasselbe wie für die erste praktische Übung: Im Falle von Fragen, die sich auf persönliche Probleme beziehen, bitte E-Mails ausschließlich an Michael Pucher, oder Gabriel Gegenhuber schreiben:

- m.pucher@univie.ac.at
- gabriel.karl.gegenhuber@univie.ac.at

Dieses mal sollte es dafür allerdings keinen Anlass geben. Im Gegensatz zur ersten Übung sollte es vom Abgabesystem mehr Feedback geben, was private Kommunikation mit uns, mit Ausnahme von Ausfällen des Abgabesystems, weitestgehend eliminieren sollte. Wir bitten darum Fragen öffentlich zu stellen, entweder im Moodle Forum, oder im inoffiziellen Uni Wien Informatik Discord[1] (dort gibt es einen Channel für Betriebssysteme).

[1] https://informatik.univie.ac.at/news-events/beitrag/news/inoffizieller-discord-server-von-studierenden-fuer-studierende-der-informatik/

2 Linux Kernel Schnittstellen

Sei es das Reservieren von Speicher, das Schreiben von Files, oder das Ausgeben von Text-Strings auf der Commandline: Alles was über "reine" Berechnung hinaus geht, muss (zumindest in einem monolithischen Kernel wie Linux) über den Betriebssystem-Kernel ausgeführt werden, da dieser die Ressourcen verwaltet. Der Linux Kernel bietet für diese Zwecke eine Vielzahl von verschiedenen Schnittstellen an und in dieser Übung werden wir uns mit zwei Schnittstellen befassen: *System calls* und *procfs*.

System calls (syscalls) sind der Hauptmechanismus um von dem unprivilegierten User-Mode in den Kernel-Mode zu wechseln, wo eine privilegierte Aktion ausgeführt werden kann. Syscalls können dabei als Funktionen einer API (Application Programming Interface) verstanden werden, analog zu den Standard-Libraries von C++ oder Java, nur eben für den Kernel. Syscalls sind prinzipiell dazu gedacht, direkt vom Maschinencode ausgeführt zu werden. Ein Beispiel: Der `write` syscall wird dazu benutzt um auf eine offene Datei zu schreiben, z.B. auf Standard Output um Text auf der Konsole auszugeben. Ein einfaches "Hello World" in `x86_64` Assembly, kann so aussehen:

```
    mov rax, 1                  # Syscall-Nummer, 1 steht für "write"
    mov rdi, 1                  # File-Deskriptor, 1 steht für stdout
    lea rsi, [rip + message]    # Der String der Ausgegeben werden soll
    mov rdx, 14                 # Die Länge des auszugebenden Strings
    syscall

message:
    .ascii "Hello, world!\n"
```

Die Kommentare im Code geben dabei an, in welchen Instruktionen welche Werte gesetzt werden, bevor der Syscall getätigt wird. Irgendwo in der letzten Abhängigkeit eines Programmes, werden syscalls in dieser Form ausgeführt, was verborgen hinter der Funktionalität von Standard-Libraries passiert. Um das ganze etwas zu vereinfachen und einige Pitfalls bei der Benützung von Syscalls zu unterbinden, gibt es die Standard-Library für die Programmiersprache C, die `libc` Bibliothek. Neben den "üblichen" Aufgaben einer Standard-Library, fungiert die `libc` daher auch als direkte System-Schnittstelle für Syscalls. Damit sieht ein Syscall in C deutlich lesbarer aus als im Assembly-Code:

```
write(1, "Hello, world!\n", 14);
```

Während es für Java oder C++ entsprechende Websites gibt, welche die Standard-Libraries dokumentieren, ist die `libc` mithilfe von den aus Aufgabe 1 bekannten `manpages` dokumentiert. Dabei ist für Syscalls vor allem Kategorie `2` relevant. Wenn man also die Dokumentation für den `write` Syscall sucht, wird man fündig, wenn man in der Commandline den Befehl `man 2 write` ausführt; selbes gilt für `man 2 uname` und alle anderen syscall Namen. Ist ein Syscall nicht über die manpages dokumentiert, hilft nur das Lesen des Linux-Kernel Source Codes und die benützung von `man 2 syscall`. Neben der Beschreibung der Syscall Argumente und Return-Werte, findet sich in den manpages auch die Information, welche Header-Dateien inkludiert werden müssen, damit das Programm kompiliert. Wenn du für die Übung C++ verwenden willst, musst du bedenken, dass es sich hierbei um C header files handelt, und du sie entsprechend mit `extern "C"` einbinden musst um potentiellen unerwarteten Problemen vorzubeugen[2], z.B.:

```
extern "C" {
#include <sys/syscall.h>
#include <unistd.h>
}
```

Das zweite relevante Interface für diese Abgabe ist das **procfs**, welches unter `man 5 proc` dokumentiert ist. Wir konzentrieren uns dabei auf den Teil, der für *Prozesse* relevant ist. Jeder Prozess hat eine PID und für jede PID gibt es einen Ordner in `/proc`, z.B. kann der Prozess mit der Nummer/PID 1234 in `/proc/1234` gefunden werden. Zusätzlich gibt es auch noch den Ordner

[2]Das sollte unter Linux nicht notwendig sein, aber better safe than sorry.

`/proc/self` der auf den Prozess zeigt, von dem dieser Ordner aus betrachtet wird. Betrachtet man einen solchen Prozess-Ordner, findet man eine Vielzahl von Dateien, welche die Eigenschaften eines Prozesses widerspiegeln:

```
total 0
dr-xr-xr-x   9 root root 0 May  8 14:41 .
dr-xr-xr-x 317 root root 0 May  8 14:41 ..
-r--r--r--   1 root root 0 May  8 19:31 arch_status
dr-xr-xr-x   2 root root 0 May  8 19:31 attr
-rw-r--r--   1 root root 0 May  8 19:31 autogroup
-r--------   1 root root 0 May  8 19:31 auxv
-r--r--r--   1 root root 0 May  8 14:42 cgroup
--w-------   1 root root 0 May  8 19:31 clear_refs
-r--r--r--   1 root root 0 May  8 14:42 cmdline
-rw-r--r--   1 root root 0 May  8 14:42 comm
-rw-r--r--   1 root root 0 May  8 19:31 coredump_filter
-r--r--r--   1 root root 0 May  8 19:31 cpu_resctrl_groups
-r--r--r--   1 root root 0 May  8 19:31 cpuset
lrwxrwxrwx   1 root root 0 May  8 19:31 cwd -> /
-r--------   1 root root 0 May  8 14:41 environ
lrwxrwxrwx   1 root root 0 May  8 14:42 exe -> /usr/lib/systemd/systemd
dr-x------   2 root root 0 May  8 14:42 fd
...
```

Beispiele für Dateien in diesem Verzeichnis sind die Commandline, mit der das Programm aufgerufen wurde (`cmdline`), ein Symlink auf das aktuelle Arbeitsverzeichnis (`cwd`), die gesetzten Environment Variablen (`environ`) und viele mehr. Im Zuge der Aufgaben gilt es herauszufinden, in welcher Datei und in welchem Format sich die gesuchten Informationen befinden.

3 `strace` als Debugging Tool

Eines der simpelsten Debugging Tools unter Linux ist `strace`. Die Aufgabe von `strace` ist es, die Syscalls aufzuzeichnen, die ein Programm während seiner Ausführung aufruft (und gegebenenfalls modifiziert). Die Syscalls werden nicht nur aufgezeichnet, sondern auch in lesbarer Form wiedergegeben, indem Zahlenwerte wieder in C Konstanten, String-Adressen in Strings, Error-Codes in die volle Fehlerbeschreibung verwandelt werden etc., z.B.:

```
openat(AT_FDCWD, "/usr/share/locale/en_US.UTF-8/LC_MESSAGES/coreutils.mo",
       O_RDONLY) = -1 ENOENT (No such file or directory)
```

Dadurch, dass ein Programm faktisch immer Syscalls ausführen muss, wenn es mit seiner Umgebung kommuniziert, lässt sich `strace` auch zum Reverse Engineering verwenden. Ein einfacher Usecase ist, z.B., herauszufinden, welche Konfigurations-Dateien ein Programm liest, falls das nicht einfach feststellbar ist. So finden sich etwa im Output von `strace mount -a` folgende Zeilen:

```
...
newfstatat(AT_FDCWD, "/etc/fstab", {st_mode=S_IFREG|0644, st_size=521, ...},
           0) = 0
openat(AT_FDCWD, "/etc/fstab", O_RDONLY|O_CLOEXEC) = 3
...
```

Sollte man nicht wissen, dass `mount -a` die Datei `/etc/fstab` liest, könnte man es mittels `strace` herausfinden. Bei der Benützung von `strace` sollte man beachten, das der initiale Output vom Executable-Loader des Betriebssystems kommt, wodurch erstmal Unmengen an Text ausgegeben werden. Kompiliert man folgendes *Hello World* Beispiel mit `g++ hello.cpp -o hello`:

```
#include <iostream>

int main() {
    std::cout << "Hello World!\n";
    return 0;
}
```

Und ruft danach `strace ./hello` auf, bekommt man etwa folgenden Output:

```
execve("./hello", ["./hello"], 0x7ffd133633f0 /* 36 vars */) = 0
brk(NULL)                               = 0x565053ae6000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffdc617e150)
    = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)
    = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
...
write(1, "Hello World!\n", 13Hello World!
)          = 13
exit_group(0)                           = ?
+++ exited with 0 +++
```

Erst in den letzten Zeilen sind die Syscalls erkennbar, die das Programm selbst ausführt. Es gehört etwas Übung dazu, zu erkennen wo der Loader endet und wo das Programm beginnt, aber grundsätzlich kann man sich im Output einfach von unten nach oben hocharbeiten. Bei dem Output an sich sollte noch beachtet werden, dass `strace` standardmäßig auf `stderr` schreibt. Mit dem `-o` Parameter kann der Output in eine beliebige Datei geschrieben werden, z.B.: `strace -o hello.log ./hello`. Ein Blick in die Manpage von `strace` (`man strace`) ist empfehlenswert.

4 Setup und Abgabe

Im anschließenden Kapitel finden sich 5 Programmieraufgaben. Für jede dieser Aufgaben ist eine Source-Code Datei abzugeben, die am Abgabe-Server kompiliert und danach gegen unsere Musterlösung getestet wird. Als Programmiersprache wird in erster Linie C++ verwendet (`*.cpp`), der Code kann aber auch als C Datei (`.c`), oder als GNU x86_64 Assembler (`*.S`) abgegeben werden. Der Code muss ohne Fehler und Warnungen kompilieren. Der Code wird nach dem `Makefile` kompiliert, das in Moodle zur Verfügung steht. Mit dem Kommando `make` werden alle Aufgaben kompiliert, einzelne Aufgaben können mittels `make $AUFGABE` kompiliert werden, z.B. `make uname` für `uname.cpp`.

Die Ausführung der Programme am Server ist stark eingeschränkt und die Programme werden während der Ausführung auf unbeabsichtigtes Verhalten beobachtet. Jede Aufgabe enthält eine Liste an erlaubten Syscalls, werden andere Syscalls aufgerufen (mit Ausnahme der Syscalls die vom Executable-Loader ausgeführt werden), wird das Programm terminiert. Die Auflistung der Syscalls enthält auch die Information, welche Syscalls notwendig sind für das Lösen der Aufgabe und welche zusätzlich vom C++ Compiler eingefügt werden. Für den Fall einer Abgabe in C oder ASM dürfen nur die notwendigen Syscalls verwendet werden.

Alle abgegebenen Programme müssen innerhalb einer Sekunde terminieren. Die Programme sollten einen return-Wert von 0 haben. Unvermeidbare Fehler sollten übersprungen werden; im Allgemeinen sollten unsere Tests allerdings keine solchen Fehler produzieren.

4.1 Setup

Um die Debian-VM der letzten Übung für diese Aufgabe zu verwenden, sollten die Pakete `build-essential` und `strace` mittels `apt` installiert werden. Das `build-essential` Meta-Paket enthält GCC, G++ und Make. Das `Makefile` in Moodle kann in den Ordner mit den eigenen Source-Files gelegt werden und ruft automatisch nach Dateiendung die richtigen Compiler-Commands auf.

4.2 Abgabe

Die Abgabe erfolgt wie schon bei der praktischen Übung 1 mithilfe eines Python-Skripts, `submit-pue2.py`. Bei der erstmaligen Ausführung des Abgabeskripts muss der persönliche Abgabetoken (siehe Feedback bei der Abgabe in Moodle) gesetzt werden.

```
$ ./submit-pue2.py reset_token
```

Neben der Abgabe der Programm Source-Codes, gibt es auch Fragen zu jedem Übungsabschnitt zu beantworten. Das Abgabeskript erstellt die dafür vorgesehenen Antwortdateien (`$AUFGABE.txt`) mittels:

```
$ ./submit-pue2.py fetch_questions
```

Nach der Fertigstellung einer Aufgabe und dem Einfügen der Antworten in die dementsprechende Antwortdatei, muss das Abgabeskript mit folgendem Befehl aufgerufen werden:

```
$ ./submit-pue2.py submit
```

Das Skript überprüft die abgegebenen Lösungen und zeigt die aktuelle Punkteübersicht an. Insgesamt können für die ganze Übung 20 Punkte (4 Punkte pro Übungsabschnitt) erreicht werden. Die Abgabe einer falschen Lösung hat keine negativen Auswirkungen. Bei mehrmaligen Abgaben zählt die Abgabe in der die meisten Punkte erreicht wurden. Für den Fall, dass Lösungen in `x84_64` Assembly geschrieben werden, gibt es die Möglichkeit, 2 Bonuspunkte pro Übungsabschnitt zu erreichen, d.h. insgesamt 10 Bonuspunkte.

5 Aufgaben

5.1 uname

- Abzugebendes Programm: `uname.cpp`

In dieser Aufgabe soll der Syscall `uname` verwendet werden (`man 2 uname`), der Versions-Informationen über den aktuell laufenden Kernel abfragen kann. Nach dem Abfragen der Informationen, sollen folgende Informationen ausgegeben werden:

- **OS:** Name des Betriebssystems
- **Hostname:** Aktueller Hostname der Maschine
- **Release:** Release-Nummer des Kernels
- **Version:** Version des Kernels (e.g. preemptive/non-preemptive, etc.)

5.1.1 Beispiel-Output

```
OS: Linux
Hostname: nyarlathotep
Release: 5.17.8-arch1-1
Version: #1 SMP PREEMPT Mon, 16 May 2022 20:45:27 +0000
```

5.1.2 Erlaubte Syscalls

Die hervorgehobenen Syscalls sind üblicherweise für die minimale Funktionalität notwendig. Die übrigen Syscalls können vom Compiler generiert werden und sind erlaubt, aber nicht direkt notwendig.

- **uname**
- **write**
- **exit/exit_group**
- newfstatat
- fstat
- futex
- fcntl

5.1.3 Fragen

- Welche Nummer (in Dezimal) hat der `uname` Syscall (unter `x86_64`)?
- In welchem Assembly-Register (unter `x86_64`) wird die Syscall Nummer gespeichert?

5.2 ps

- Abzugebendes Programm: `ps.cpp`

In dieser Aufgabe soll das `procfs` verwendet werden, um eine simple Variante des `ps` Linux-Tools nachzubauen. Es ist erlaubt sich dazu den Source Code von verschiedenen `ps` Implementierungen anzusehen, allerdings nicht erlaubt, diesen Code exakt zu kopieren (was in diesem Fall auch viel komplexer sein sollte, als den Code selbst zu schreiben). Alle notwendigen Informationen sollten sich in `man 5 proc` finden lassen.

Wie das richtige `ps`, soll das Programm über alle Prozesse iterieren und dabei Informationen sammeln. Sollte man auf eine der relevanten Informationen nicht zugreifen können (e.g. `Permission denied` beim Öffnen einer relevanten Datei), soll der Prozess im Output ignoriert werden. Folgende Informationen sollen gesammelt und im JSON Format (siehe Beispiel) ausgegeben werden:

- **pid:** Die PID des Prozesses
- **exe:** Der Pfad zum Executable File des Prozesses
- **cwd:** Das aktuelle Arbeitsverzeichnis des Prozesses
- **base_address:** Die Basisadresse des Executable Files im Speicher (Dezimal)
- **state:** Der aktuelle Prozessstatus
- **cmdline:** Die vollständige Aufrufzeile des Prozesses, als Array

5.2.1 Beispiel-Output

Hinweis: Das Beispiel ist pretty-printed JSON. Grundsätzlich ist es egal wie das JSON ausgegeben wird, am besten sollte es komplett ohne Whitespace und Linebreaks ausgegeben werden. Zur Kontrolle kann der Output durch `jq` gepiped werden (`./ps | jq`), einem Commandline JSON Processor, welcher per Default pretty-printed JSON ausgibt. Der Grading-Server wird `jq 'sort_by(.pid)' --sort-keys` verwenden, um den JSON Output zu normalisieren.

```
[
  {
    "pid": 841,
    "exe": "/usr/lib/systemd/systemd",
    "cwd": "/",
    "base_address": 93990369611776,
    "state": "S",
    "cmdline": [
      "/usr/lib/systemd/systemd",
      "--user"
    ]
  },
  {
    "pid": 849,
    "exe": "/usr/bin/bash",
    "cwd": "/home/cluosh",
    "base_address": 93883351433216,
    "state": "S",
    "cmdline": [
      "/bin/sh",
      "/usr/bin/startx",
      "--",
      "-keeptty"
    ]
  },
 ...
]
```

5.2.2 Erlaubte Syscalls

Die hervorgehobenen Syscalls sind üblicherweise für die minimale Funktionalität notwendig. Die übrigen Syscalls können vom Compiler generiert werden und sind erlaubt, aber nicht direkt notwendig.

- **getdents64**
- **openat**
- **readlink**
- **read**
- **write**
- **close**
- **exit/exit_group**
- newfstatat
- fstat
- futex
- fcntl
- stat
- lstat
- brk

5.2.3 Fragen

- Welcher Syscall wird verwendet um durch ein Verzeichnis zu iterieren?
- Welche Datei im procfs Ordner eines Prozesses gibt Aufschluss auf die Adressbereiche, die ein Prozess verwendet?
- Was bedeutet der Prozess-Status **D** (englische Bezeichnung aus `man ps`)?

5.3 pstree

- Abzugebendes Programm: `pstree.cpp`

Diese Aufgabe ist verwendet ähnliche Grundbausteine wie `ps` und man kann Code wiederverwenden. Es lohnt sich daher die `ps` Aufgabe zuerst zu machen, oder als Anhaltspunkt zu verwenden.

Prozesse können Child-Prozesse besitzen und das `pstree` Kommando erlaubt die Auflistung von Parent/Child-Prozessen in Baumform. In dieser Aufgabe sollen alle Prozesse und ihre Child-Prozesse hierarchisch aufgelistet werden, wieder mittels JSON output. Jede Node im Baum sollte folgende Informationen beinhalten:

- **pid:** Die PID des Prozesses
- **name:** Der Name des Prozesses
- **children:** Array mit den Kindern des Prozesses

5.3.1 Beispiel-Output

```
[
  {
    "pid": 1,
    "name": "systemd",
    "children": [
      {
        "pid": 413,
        "name": "systemd-journal",
        "children": []
      },
      {
        "pid": 429,
        "name": "systemd-udevd",
        "children": []
      },
      {
        "pid": 439,
        "name": "systemd-network",
        "children": []
      },
      ...
    ]
  }
]
```

5.3.2 Erlaubte Syscalls

Die hervorgehobenen Syscalls sind üblicherweise für die minimale Funktionalität notwendig. Die übrigen Syscalls können vom Compiler generiert werden und sind erlaubt, aber nicht direkt notwendig.

- **getdents64**
- **openat**
- **read**
- **write**
- **close**
- **exit/exit_group**
- newfstatat
- fstat
- futex
- fcntl

- stat
- brk

5.3.3 Fragen

- Wie wird der Parent-Prozess (fast) aller Prozesse unter Linux üblicherweise genannt?
- Welche PID hat dieser Prozess?

Zusätzlicher Denkanstoß (wird nicht bewertet): Gibt es Prozesse, die keine Nachfahren des gesuchten Prozesses sind? Welche?

5.4 killall

- Abzugebendes Programm: `killall.cpp`

Diese Aufgabe ist verwendet ähnliche Grundbausteine wie `ps` und man kann Code wiederverwenden. Es lohnt sich daher die `ps` Aufgabe zuerst zu machen, oder als Anhaltspunkt zu verwenden.

In der ersten praktischen Übung wurde das `killall` Programm verwendet und in dieser Aufgabe geht es darum, eine einfache Version dieses Programms nachzubauen. Es wird ein String als erster Commandline-Parameter übergeben (`argv[1]`). Alle Prozesse, die diesen String im Namen enthalten, sollen mittels dem `kill` Syscall (`man 2 kill`) und dem `SIGKILL` Signal terminiert werden.

5.4.1 Beispiel-Output

Dieses Programm soll keinen Output produzieren. Beispiel-Aufruf:

```
./killall Discord
```

5.4.2 Erlaubte Syscalls

Die hervorgehobenen Syscalls sind üblicherweise für die minimale Funktionalität notwendig. Die übrigen Syscalls können vom Compiler generiert werden und sind erlaubt, aber nicht direkt notwendig.

- **kill**
- **getdents64**
- **openat**
- **read**
- **close**
- **exit/exit_group**
- newfstatat
- fstat
- futex
- fcntl
- stat
- brk
- write (Für Debugging-Output)

5.4.3 Fragen

- Wie heißt das Signal mit der Nummer 14?

5.5 statx

- Abzugebendes Programm: `statx.cpp`

In dieser Aufgabe soll ein relativ neuer Syscall namens `statx` (`man 2 statx`) verwendet werden (hinzugefügt in Linux 4.11). Dieser Syscall erlaubt das Bestimmen von Datei-Attributen, wie Berechtigungen, Größe, etc., und im Gegensatz zu den alten `stat` Syscalls genauere Kontrolle darüber, welche Informationen abgefragt werden. Es wird ein absoluter Pfad als erster Commandline-Parameter übergeben (`argv[1]`). Der `statx` Syscall soll **drei mal** mit unterschiedlichen Parametern aufgerufen werden (ohne Symbolic Links aufzulösen). Es sollen dabei immer nur die Attribute abgefragt werden, die wirklich benötigt werden:

1. **Berechtigungen:** Beim ersten `statx` Aufruf sollen nur die Berechtigungen abgefragt werden, und dabei einfach nur *Read*, *Write*, *Execute* für *Owner*, *Group* und *Other*. Diese Berechtigungen sollen dann wie bei `ls` ausgegeben werden, z.B. `rwxr-xr-x`.
2. **UID/GID:** Beim zweiten `statx` Aufruf sollen gleichzeitig die UID des Owners und die GID der Gruppe der Datei abgefragt werden. Die IDs sollen dann so ausgegeben werden: `UID: 1000, GID: 1000`
3. **Dateigröße:** Beim letzten `statx` Aufruf soll die Dateigröße in Bytes abgefragt und in Dezimal ausgegeben werden, z.B. `39503`.

5.5.1 Beispiel-Output

Beispiel-Output für `./statx /bin/ls` (Output wird je nach System entsprechend anders sein):

```
rwxr-xr-x
UID: 0, GID: 0
Size: 137744
```

5.5.2 Erlaubte Syscalls

- **statx**
- **write**
- **exit/exit_group**
- newfstatat
- fstat
- futex
- fcntl

5.5.3 Fragen

- Welche Kombination von `statx` mask Konstanten ergibt den Wert `0x8a8`? (Tipp: `strace`)