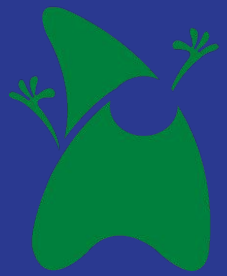
Vom Java-Entwickler zum Low-Level-Hacker

Persönliche Erfahrungen, Hürden, Probleme und Lösungen

Philipp Schuster / @phip1611
23. September 2022



JUG SAXONY DAY

Agenda

1. Einleitung 5 min
   - 1.1. Motivation des Talks
   - 1.2. Über Mich
   - 1.3. Cyberus Technology
2. “Low-Level-Entwicklung ist hässlich” 10 min
3. Produktiv Low-Level-Code schreiben 15 min
   - 3.1. Notwendige Kenntnisse
   - 3.2. Rust als “New Player in the Game”
4. Praktisches Beispiel: Roottask für Hedron 15 min
5. Fazit & Fragerunde 15 min

Agenda

1. Einleitung 5 min
   - 1.1. Motivation des Talks
   - 1.2. Über Mich
   - 1.3. Cyberus Technology
2. “Low-Level-Entwicklung ist hässlich” 10 min
3. Produktiv Low-Level-Code schreiben 15 min
   - 3.1. Notwendige Kenntnisse
   - 3.2. Rust als “New Player in the Game”
4. Praktisches Beispiel: Roottask für Hedron 15 min
5. Fazit & Fragerunde 15 min

Agenda

1. Einleitung 5 min
   - 1.1. Motivation des Talks
   - 1.2. Über Mich
   - 1.3. Cyberus Technology
2. “Low-Level-Entwicklung ist hässlich” 10 min
3. Produktiv Low-Level-Code schreiben 15 min
   - 3.1. Notwendige Kenntnisse
   - 3.2. Rust als “New Player in the Game”
4. Praktisches Beispiel: Roottask für Hedron 15 min
5. Fazit & Fragerunde 15 min

Agenda

1. Einleitung 5 min
   - 1.1. Motivation des Talks
   - 1.2. Über Mich
   - 1.3. Cyberus Technology
2. “Low-Level-Entwicklung ist hässlich” 10 min
3. Produktiv Low-Level-Code schreiben 15 min
   - 3.1. Notwendige Kenntnisse
   - 3.2. Rust als “New Player in the Game”
4. Praktisches Beispiel: Roottask für Hedron 15 min
5. Fazit & Fragerunde 15 min

Agenda

1. Einleitung 5 min
   - 1.1. Motivation des Talks
   - 1.2. Über Mich
   - 1.3. Cyberus Technology
2. “Low-Level-Entwicklung ist hässlich” 10 min
3. Produktiv Low-Level-Code schreiben 15 min
   - 3.1. Notwendige Kenntnisse
   - 3.2. Rust als “New Player in the Game”
4. Praktisches Beispiel: Roottask für Hedron 15 min
5. Fazit & Fragerunde 15 min

1. Einleitung

1.1 Motivation des Talks

- Java/Spring Boot-Anwendung

  → Container → Kubernetes → Millionen CPU-Zyklen

  → cool und produktiv, aber: Was passiert eigentlich auf der Hardware?
- Hinweise geben für Einstieg in Low-Level-Programmierung

  + relevante Technologien und Techniken
- aktuell ist sehr gute Zeit um in Low-Level einzusteigen, da dank Rust viel Bewegung ins Ökosystem kommt

1.2 Über mich

- Philipp Schuster, 25 Jahre
- Software-Engineer bei Cyberus Technology GmbH

  Betriebssysteme, Low-Level-Programmierung
- Kontakt:
  - philipp.schuster@phip1611.de
  - @phip1611
  - https://phip1611.de

1.2 Über mich

- von 2016 – 2021 in T-Systems MMS

  Frontend: Angular, Typescript, CSS

  Backend: Java, Spring Boot
- Ab 2017 in Uni fast alle Wahlpflichtmodule im Bereich Betriebssysteme
- zeitgleich privat versucht, mit C/C++ mehr und mehr zu programmieren
- ab 2019 mir selbst Rust beigebracht + viele private Projekte
- Großer Beleg (2020/21) im Bereich Betriebssysteme

1.2 Über mich

CYBERUS TECHNOLOGY

- seit 05/2021 bei Cyberus Technology GmbH als Werkstudent
- von 09/2021 – 04/2022: Diplomarbeit

  “A Policy-free System-Call Layer for the Hedron Microhypervisor”

  https://github.com/phip1611/diplomarbeit-impl
- seit 06/2022 – Vollzeit angestellt

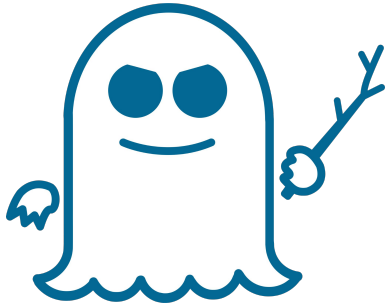
1.2 Über mich

- viel Open Source-Arbeit auf GitHub
- Teil des “rust-osdev”-Teams auf GitHub
- diverse eigene Rust-Bibliotheken auf crates.io
- Contributions zu zahlreichen dritten Rust-Projekten
- eine Java-Bibliothek auf Maven Central

1.3 Arbeit bei Cyberus Technology

CYBERUS
TECHNOLOGY

- 02/2017 in Dresden gegründet
- aktuell ~21 Mitarbeiter:innen
- bekannt u.A. durch *Meltdown* und *Spectre*:
  Mitarbeiter stehen auf den Papern



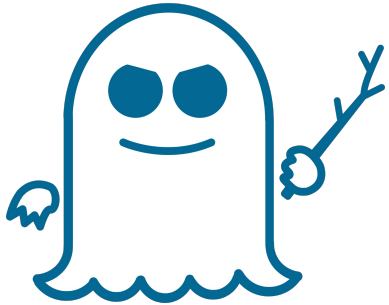
1.3 Arbeit bei Cyberus Technology

CYBERUS
TECHNOLOGY

Dresdner an Entdeckung der Intel-Sicherheitslücke beteiligt

Die Software-Experten der Cyberus Technology GmbH halten Updates nicht für ausreichend

Teilen | Folgen



1.3 Arbeit bei Cyberus Technology

CYBERUS TECHNOLOGY

- Hauptprodukt: **S**ecure **V**irtualization **P**latform (SVP)
  - Basis: Mikrokern Hedron + Runtime Environment

    → eigenes, kleines Betriebssystem
  - sichere, performante virtuelle Maschinen erstellen

    → GPU-Virtualisierung auf alten und neuen Intel-Generationen
  - sicherer als Linux/KVM und Microsoft/HyperV

    → Mikrokernarchitektur

1.3 Arbeit bei Cyberus Technology

CYBERUS TECHNOLOGY

- Großer Fokus auf Testen von Kernel + Runtime Environment auf echter Hardware: Internes Produkt “SoTest”
- jede Änderung
  → GitLab CI an “SoTest” angebunden
  → komplette Testung auf verschiedener Hardware

1.3 Arbeit bei Cyberus Technology

CYBERUS TECHNOLOGY

- Hedron (Kernel) ist quelloffen:
  https://github.com/cyberus-technology/hedron
- stammt von NOVA ab → TU Dresden 2010

2. “Low-Level-Entwicklung ist hässlich”

2. “Low-Level-Entwicklung ist hässlich”

- “Low-Level-Entwicklung ist hässlich”

  → oft gehörter Satz
- Low-Level-Software:
  - Kernel
  - Treiber
  - Firmware
  - Embedded Systems

2. “Low-Level-Entwicklung ist hässlich”

Hintergrundwissen: Wie baut man Low-Level-Projekte*?

- Sprachen: C/C++ & Assembly
- Compiler: GCC
- Build-System: GNU Make, GNU Automake, CMake

* Gilt für die Vielzahl wichtiger, jahrzehntealter und bis heute lebender Low-Level-Projekte

2. “Low-Level-Entwicklung ist hässlich”

Was ist schöner Code?

- einheitliche Formatierung durch Formatter
- einheitlicher Codestyle durch Linter
- Kommentare am Code

2. “Low-Level-Entwicklung ist hässlich”

Was ist schöner Code?

- einheitliche Formatierung durch Formatter
- einheitlicher Codestyle durch Linter
- Kommentare am Code
- Variablennamen aus einem Buchstaben vermeiden
- Funktionsnamen und Variablen sollten sprechend und passend sein
- Änderungen leicht durchführbar, ohne etwas kaputtzumachen (silent bug)

2. “Low-Level-Entwicklung ist hässlich”

Fun Facts

- Kommentar “this is ugly” 🤡

  → 12x im Linux Source Code (5.19)

  → 0x in latest Spring Boot (2.7)

- Kommentar “wtf”/”fucked up” 🤡

  → 21x im Linux Source Code (5.19)

  → 0x in latest Spring Boot (2.7)

```
case 0x02: /* HDMI */
        type = DCB_CONNECTOR_HDMI_1;
        break;
case 0x03: /* DVI-D */
        type = DCB_CONNECTOR_DVI_D;
        break;
case 0x0e: /* eDP, falls through to DPint */
        ctx.outp[1] |= 0x00010000;
        fallthrough;
case 0x07: /* DP internal, wtf is this?? HP8670w */   Skeggs, 22.07.12
        ctx.outp[1] |= 0x00000004; /* use_power_scripts? */
        type = DCB_CONNECTOR_eDP;
        break;
default:
```

Linux @ 5.19.9: drivers/gpu/drm/nouveau/nvkm/subdev/mxm/nv50.c

```
    struct pollfd pfd[3] = { {.fd=0}, {.fd=1}, {.fd=2} };
    int r =
#ifdef SYS_poll
    __syscall(SYS_poll, pfd, 3, 0);
#else
    __syscall(SYS_ppoll, pfd, 3, &(struct timespec){0}, 0, _NSIG/8);
#endif
    if (r<0) a_crash();
    for (i=0; i<3; i++) if (pfd[i].revents&POLLNVAL)
        if (__sys_open("/dev/null", O_RDWR)<0)
            a_crash();
    libc.secure = 1;
```

musl/libc @ 1.2.3: src/env/__libc_start_main.c

```
/*
 * System-Call Entry
 */
.align                  4, 0x90
.globl                  entry_sys
entry_sys:
#if defined(__CET__) && (__CET__ & 1)
                        endbr64
#endif
#if defined(__CET__) && (__CET__ & 2)
                        PATCH   (setssbsy,,PATCH_CET_SS)
#endif
                        mov     %rsp, %r11
                        mov     tss_run + 4, %rsp
                        lea     -(__SIZEOF_POINTER__ * 8)(%rsp), %rsp
                        SAVE_GPR
                        lea     DSTK_TOP, %rsp
```

NOVA Microhypervisor @ release-22.35.0: src/x86_64/entry.S

2. “Low-Level-Entwicklung ist hässlich”

- viele wichtige Low-Level-Projekte haben schwer lesbaren, unintuitiven Code
- gibt es auch in der High-Level-Welt, aber:

2. “Low-Level-Entwicklung ist hässlich”

Tooling für “schönen” Code

- in Webentwicklung/High-Level vielfach mehr Entwickler:innen
  - neue Impulse und Dinge “neu denken”
  - Projekte weniger langlebig bzw. es werden oft neue gestartet
  - Tooling für Linting, Formatting etc. ausgereift und etabliert

2. “Low-Level-Entwicklung ist hässlich”

Tooling für “schönen” Code

- In Low-Level-Welt deutlich weniger Entwickler:innen
  - viele langlebige Projekte von eingefleischten Maintainern
  - wenig neue Impulse
  - Änderungen im Ökosystem vergleichsweise langsam

2. “Low-Level-Entwicklung ist hässlich”

Tooling für “schönen” Code

- Java: Checkstyle als Linter und Formatter
- JavaScript und Typescript: ESLint als Linter, Prettier als Formatter

→ einfach zu nutzendes Tooling; in vielen “Starterprojekten” dabei

2. “Low-Level-Entwicklung ist hässlich”

Tooling für “schönen” Code

- in Make- oder CMake-basierten Projekten müssen Linter und Formatter manuell ins Build-System eingebaut werden
- gibt keine etablierten (CLI-)Tools, die einem ein Starterprojekt generieren, (anders als bei Angular, Vue, Spring Boot, ...)

- gibt allerdings für C/C++ Clang Tidy als Linter und Clang Format als Formatter

- Rust macht das übrigens alles besser 🥳 (später mehr)

2. “Low-Level-Entwicklung ist hässlich”

Tooling für “schönen” Code

- Erfahrung/Beobachtung zeigt, dass in etablierte, alte Code-Basen von C/C++-Projekten selten moderne Formatter und Linter eingebaut werden
  → GRUB, musl/libc, ...
- demzufolge auch kein Zwang zu Codedokumentation, da es keine CI-Steps dafür gibt

2. “Low-Level-Entwicklung ist hässlich”

Testen ist aufwändiger

- Testen von Webseiten oder Webservices im Vergleich relativ einfach

  → Prozess starten, Ergebnis anschauen

  → einfache Automatisierung mit Unit- und Integrationstests
- UI-Entwicklung mit Live-Reload
- Low-Level Code: In VM testen

  → benötigt bootbare Datei

2. “Low-Level-Entwicklung ist hässlich”

Ist das wirklich so? Nicht zwingend!

- “Low-Level-Entwicklung ist hässlich”

  → Low-Level-Welt ist dominiert von alten, wichtigen Projekten

  → Satz trifft in einigen Dimensionen daher zu

  → für neue Projekte gilt das nicht, weil man sich mit vorhandemem Tooling “schöne” Projekte aufsetzen kann

- die Zahnräder drehen sich langsam in langlebigen Projekten

2. “Low-Level-Entwicklung ist hässlich”

Ist das wirklich so? Nicht zwingend!

- im Ökosystem von C/C++ (und Assembly) ist es einfach “schlechten” Code ohne Kommentare zu schreiben
- gibt aber auch dafür heute gutes Tooling
- Rust und sein Ökosystem beseitigen viele Probleme, die die jahrzehntelange Erfahrung mit C/C++ und CMake/Make offenbart haben

2. “Low-Level-Entwicklung ist hässlich”

Persönliche Motivation

- mich persönlich fasziniert die Welt
- möchte verstehen, wie moderne IT-Systeme im Kern funktionieren
- ich habe Freude daran gut kommentierte, minimale Beispiele zu bauen (siehe Blog oder GitHub)

3. Produktiv Low-Level-Code schreiben

3.1 Notwendige Kenntnisse

Sprachen + deren Buildsysteme

- C/C++
- Assembly (= Sprache; Assembler: Konverter zu Maschinensprache)
- im zweiten Schritt auch Rust (sowie Interoperabilität mit C/C++)
- Linker Scripte ←→ ELF / Executable Files

3.1 Notwendige Kenntnisse

Compiler-Output lesen

C:

```c
long mult(long a, long b)
{
    return a * b;
}
```

Assembly:

```
mult:   mov     rax, rdi
        imul    rax, rsi
        ret
```

3.1 Notwendige Kenntnisse

Assembly-Code in High-Level-Code springen

Assembly:

```
# prepare first arg
mov rdi, 0x1234000
# prepare second arg
mov rsi, 7
jmp rust_entry
```

Rust:

```
#[no_mangle]
fn rust_entry(
    str_ptr: *const u8,
    str_len: u64
) {
    /* ... */
}
```

3.1 Notwendige Kenntnisse

Was produziert der Linker?

- man kann mit C oder Rust sowohl Webserver als auch Kernel programmieren:

  Was ist der technische Unterschied im Kompilat/Binary?

  → “standard binary” vs. “no-standard/freestanding binary”

3.1 Notwendige Kenntnisse

Was ist ein “standard binary”?

- Beispiel: Apache, Java-Runtime, PHP Interpreter, Webbrowser

  → gelinkt gegen Standardbibliothek, wie libc
- Standardbibliothek bindet typische Funktionen (Dateien, Netzwerk) via Syscalls an die Kernel-API
- “Standard-Binary” besteht aus Applikations-Code plus Standardbibliothek

3.1 Notwendige Kenntnisse

Was ist ein “standard binary”?

- C-Programme nutzen direkt libc: open(), write()
- Go/Rust/Java haben auf Linux Bindings von ihrer Standardbibliothek auf die libc
- analog auf MacOS und Windows

3.1 Notwendige Kenntnisse

Was ist ein “no-standard/freestanding binary”?

- keine Standardbibliothek
- keine out-of-the-box Speicherverwaltung
- keine out-of-the-box Dateisystemabstraktion

→ kein Runtime Environment, weil Kernel und einige no-std Anwendungen das Runtime Environment sind/bilden

- “no-standard-Binary” besteht nur aus Applikations-Code

3.1 Notwendige Kenntnisse

Wie testet man einen Kernel?

- allgemeine Funktionalität in Bibliothek auslagern
- Code testbar schreiben (zum Beispiel über generische Adapter)
- auf Hostsystem mittels Unittests testen
- Kernel/Firmware Binary

  → starten in einer VM, bspw. mittels **QEMU** (https://www.qemu.org/)
- für Integrationstest kann ein dediziertes Binary gebaut werden

3.1 Notwendige Kenntnisse

Wie kommuniziert der Kernel nach außen?

- Daten in ein Register schreiben, CPU stoppen, mit QEMU auslesen
- Byte-weise Daten (ASCII) rausschreiben über (virtuelles) Serial-Gerät
- QEMU kann mit einem speziellen Exit-Code beendet werden

  → Testfehlschläge können so mitgeteilt werden
- bei genügend Initialisierung irgendwann auch Netzwerk, Display oder Sound nutzbar

3.1 Notwendige Kenntnisse

Wie greift man auf Hardware zu?

- Hardware ist immer auf den physischen Adressraum gemappt
- nicht alle physischen Adressen entsprechen dem RAM
- auf x86 gibt es zusätzlich I/O Ports, die auf Hardware-Geräte mappen

3.1 Notwendige Kenntnisse

Lernressourcen

- Spezifikationen lesen (UEFI, Multiboot2, ...)
- allgemeine Hilfestellungen: https://wiki.osdev.org
- **Minimalprojekte bauen** (“Minimalkernel”, der “Hello World” ausgibt)
- Von Open Source-Lösungen lernen und dann adaptieren

  → aber Code leider oft “hässlich” und schwer zu verstehen;

  insbesondere Assembly ohne Kommentare
- Geduld mitbringen und erfahrene Entwickler:innen fragen

3.2 Rust als “New Player in the Game”

Was ist Rust?

- Rust seit 2015 populär geworden
- neue Projekte nutzen Rust schon einigen Jahren
- wird mittel- und langfristig zahlreiche C/C++-Projekte ersetzen oder ergänzen
- sehr bald im Linux-Kernel unterstützt, um neue Treiber zu schreiben

- <rust-propaganda>Rust ist auch gut geeignet für asynchrone Programmierung und Web-Services</rust-propaganda>

3.2 Rust als “New Player in the Game”

Was macht Rust besonders?

- Rusts größter Beitrag zur Softwareentwicklung ist die Unifizierung
  - **des Buildsystems, das out-of-the-box funktioniert (Cargo),**
  - der Codeformatierung,
  - von Linting-Regeln,
  - Fehlerbehandlung
  - und der Codedokumentation + generierung einer HTML-Version

- alles über Cargo durchführbar

3.2 Rust als “New Player in the Game”

Was macht Rust besonders?

- in Kombination mit exzellenter Standardbibliothek, High-Level-Abstraktionen und sehr effektivem Compiler (viele Abstraktionen sind zero-cost)
- “libcore” ist Plattform-unabhängiger Teil der Standardbilbiothek (“libstd”), der auch für “no-std binaries” genutzt werden kann

  → lesende String-Operationen

  → Iteratoren

  → Pointer-Arithmetik

  → Formatierstrings...

4. Praktisches Beispiel: Roottask für Hedron

4. Praktisches Beispiel: Roottask für Hedron

https://github.com/phip1611/hedron-minimal-roottask

4. Praktisches Beispiel: Roottask für Hedron

- Hedron ist Mikrokern, der auf x86_64 läuft
- Hedron ist nicht Linux/UNIX: eigene Konzepte → siehe NOVA Paper von 2010
- kennt Prozesse mit mehreren Threads

4. Praktisches Beispiel: Roottask für Hedron

- zeige minimales Setup für eine Roottask
- Hedron bekommt Roottask als Bootmodul (BLOB im Speicher) mit

  → lädt ausführbare ELF-Datei in den Speicher

  → Ausführung beginnt

4. Praktisches Beispiel: Roottask für Hedron

- bauen ein ELF mit Rust/Cargo
- Kompilieren ein “no-std binary” x86_64 (“bare metal”)
- ein wenig Assembly-Code ist notwendig für Start-Routine
- eigenes Linker-Script beschreibt, wie das ELF zusammen gebaut werden soll
- ELF nicht lauffähig unter Linux, nur unter Hedron

5. Fazit & Fragerunde

5. Fazit

Ist Low-Level “hässlich”?

- Altprojekte sind es, wie in jedem Ökosystem
- Low-Level-Welt ist dominiert von sehr alten, immer noch genutzten Projekten (Linux, musl/libc und glibc, Bash-Shell, ...)

5. Fazit

Ist Low-Level “hässlich”?

- Altprojekte sind es, wie in jedem Ökosystem
- Low-Level-Welt ist dominiert von sehr alten, immer noch genutzten Projekten (Linux, musl/libc und glibc, Bash-Shell, ...)
- nicht abschrecken lassen, weil man selbst modernes Tooling nehmen kann
- Low-Level ist nicht magisch. Man lernt es genau wie man jede andere Nische kennenlernt.
- Rust ist gerade massiver Beschleuniger für zahlreiche neue Low-Level-Projekte → gute Zeit um einzusteigen!

Interessante Links

- NOVA-Paper: https://hypervisor.org/eurosys2010.pdf
- Hedron-Quellcode: https://github.com/cyberus-technology/hedron
- Low-Level-Projekte von mir (mit Fokus auf guter Dokumentation):
  - **https://github.com/phip1611/hedron-minimal-roottask**
  - https://github.com/phip1611/multiboot2-binary-rust
  - https://github.com/phip1611/rust-different-calling-conventions-example
  - https://github.com/phip1611/direct-syscalls-linux-from-rust