Stefan Walter + Claude

SySS
THE PENTEST EXPERTS.

Tracking Linux Syscalls

From Disassembly to eBPF



WHOAMI

- Seit 2021 bei der SySS GmbH in Tübingen
- Hauptberuflich: Sicherheit interner Unternehmensnetzwerke, Fokus Windows / Active Directory
- In der Freizeit breit technisch interessiert – und natürlich auch an Linux
- Dieser Vortrag: ursprünglich ein Freizeit-Experiment, aufbereitet

WAS MACHT EIN LINUX-SYSTEM EIGENTLICH?

- Ein Experiment
- Abstraktionsebene hier: **Syscalls**
- Interaktion von Anwendungen mit Hardware / Außenwelt geht über Kernel

SYSCALLS

WAS IST EIN SYSCALL?

- Kontrollierter Übergang von **User Mode** $\Rightarrow$ **Kernel Mode**
- Interaktion mit Hardware, privilegierte Operationen

  Dateien, Netzwerk, Prozesse, Speicher, Zeit, …
- Im Linux-Kernel: stabile Schnittstelle
- x86-64 Linux: < 500 Syscalls

Wer definiert was?

Beteiligte Komponenten:

- Hardware (hier Fokus nur auf: amd64/x86-64-Prozessoren)
- Software / Kernel (hier Fokus nur auf: Linux-Kernel)
- User-Space Software / Applikationen / Standardbibliotheken (hier: Spezialfall glibc)

Gute Quelle: https://blog.packagecloud.io/the-definitive-guide-to-linux-system-calls/

Hardware

x86-64 Prozessoren: Hardware Contract

https://docs.amd.com/v/u/en-US/24593_3.44_APM_Vol2
AMD64 Architecture Programmer's Manual Volume 2: System Programming (PUB) (24593)

- Kapitel 6 System Instructions (MSRs)

Historisch verschiedene Syscall-Methoden:

- Kapitel 6.1.2 SYSENTER and SYSEXIT (Legacy Mode Only)
- Kapitel 6.1.1 SYSCALL and SYSRET (Long Mode / 64-Bit Mode)
- (Linux-OS-spezifisch: per Interrupt mit `int 0x80`)

SYSCALL und SYSRET (1/2)

- Typischer moderner Weg: SYSCALL/SYSRET-Assembler-Instruktion

https://docs.amd.com/v/u/en-US/24594_3.37
AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Programming Instructions (PUB) (24594)

- Seite 483ff., System Instruction Reference, SYSCALL
- Seite 490ff., System Instruction Reference, SYSRET

SYSCALL und SYSRET (2/2)

Entsprechende Abschnitte in der Intel-Dokumentation:

- https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z

Instruktion SYSCALL Spec

AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Programming Instructions (PUB) (24594)

SYSCALL in Long Mode 64-bit mode:

- transfers control to fixed entry point in an OS
- saves RIP of following instruction into RCX
- loads new RIP from LSTAR bits 63:0 (MSR 0xC0000082)
- sets CPL to 0

Instruktion SYSRET Spec

- Can only be executed in CPL 0
- Sets CPL to 3
- Loads RIP from RCX

Hinweis zu den Demos

Für alle praktischen Teile des Talks: Hostsystem:

- Debian 13 (Trixie) Linux
- x86_64 CPU: 12th Gen Intel® Core™ i7-1270P

Syscall-Kernel-Handler im (ro)-Debugger

```
sudo apt install linux-image-$(uname -r) msr-tools gdb
sudo modprobe msr

sudo rdmsr 0xC0000082
sudo grep -w "entry_SYSCALL_64" /proc/kallsyms

sudo gdb -q /usr/lib/debug/vmlinux-$(uname -r) /proc/kcore
x/40i &entry_SYSCALL_64
info line entry_SYSCALL_64
```

Software

Syscall-Kernel-Handler im Quellcode (1/7)

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git --branch v7.1 --depth=1
```

Syscall-Kernel-Handler im Quellcode (2/7)

Setup Syscall-Handler beim Boot: `arch/x86/kernel/cpu/common.c`

```
void syscall_init(void)
{
    /* The default user and kernel segments */
    wrmsr(MSR_STAR, 0, (__USER32_CS << 16) | __KERNEL_CS);
        ...
        idt_syscall_init();
}

static inline void idt_syscall_init(void)
{
    wrmsrq(MSR_LSTAR, (unsigned long)entry_SYSCALL_64);

    if (ia32_enabled()) {
        wrmsrq_cstar((unsigned long)entry_SYSCALL_compat); ...
    } else {
        wrmsrq_cstar((unsigned long)entry_SYSCALL32_ignore); ...
    }
    ...
}
```

Syscall-Kernel-Handler im Quellcode (3/7)

Syscall-Handler: `arch/x86/entry/entry_64.S`

```
SYM_CODE_START(entry_SYSCALL_64)
    UNWIND_HINT_ENTRY
    ENDBR

    swapgs
    /* tss.sp2 is scratch space. */
    movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
    SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
    movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

    ...

    call    do_syscall_64       /* returns with IRQs disabled */
```

Syscall-Kernel-Handler im Quellcode (4/7)

`arch/x86/entry/syscall_64.c`

```
static __always_inline bool do_syscall_x64(struct pt_regs *regs, int nr)
{
    /*
     * Convert negative numbers to very high and thus out of range
     * numbers for comparisons.
     */
    unsigned int unr = nr;

    if (likely(unr < NR_syscalls)) {
        unr = array_index_nospec(unr, NR_syscalls);
        regs->ax = x64_sys_call(regs, unr);
        return true;
    }
    return false;
}
```

Syscall-Kernel-Handler im Quellcode (5/7)

```
long x64_sys_call(const struct pt_regs *regs, unsigned int nr)
{
    switch (nr) {
    #include <asm/syscalls_64.h>
    default: return __x64_sys_ni_syscall(regs);
    }
}
```

Syscall-Kernel-Handler im Quellcode (6/7)

`asm/syscalls_64.h` ist automatisch generiert

`arch/x86/entry/syscalls/Makefile`

```
syscall32 := $(src)/syscall_32.tbl
syscall64 := $(src)/syscall_64.tbl

syshdr := $(srctree)/scripts/syscallhdr.sh
systbl := $(srctree)/scripts/syscalltbl.sh

...

$(out)/syscalls_64.h: abis := common,64
$(out)/syscalls_64.h: $(syscall64) $(systbl) FORCE
    $(call if_changed,systbl)

...
```

Syscall-Kernel-Handler im Quellcode (7/7)

`arch/x86/entry/syscalls/syscall_64.tbl`

```
...
# 64-bit system call numbers and entry vectors
...
# The format is:
# <number> <abi> <name> <entry point> [<compat entry point> [noreturn]]
...
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
6       common  lstat                   sys_newlstat
7       common  poll                    sys_poll
8       common  lseek                   sys_lseek
9       common  mmap                    sys_mmap
10      common  mprotect                sys_mprotect
...
```

Sysret im Kernel Quellcode

`arch/x86/entry/entry_64.S`

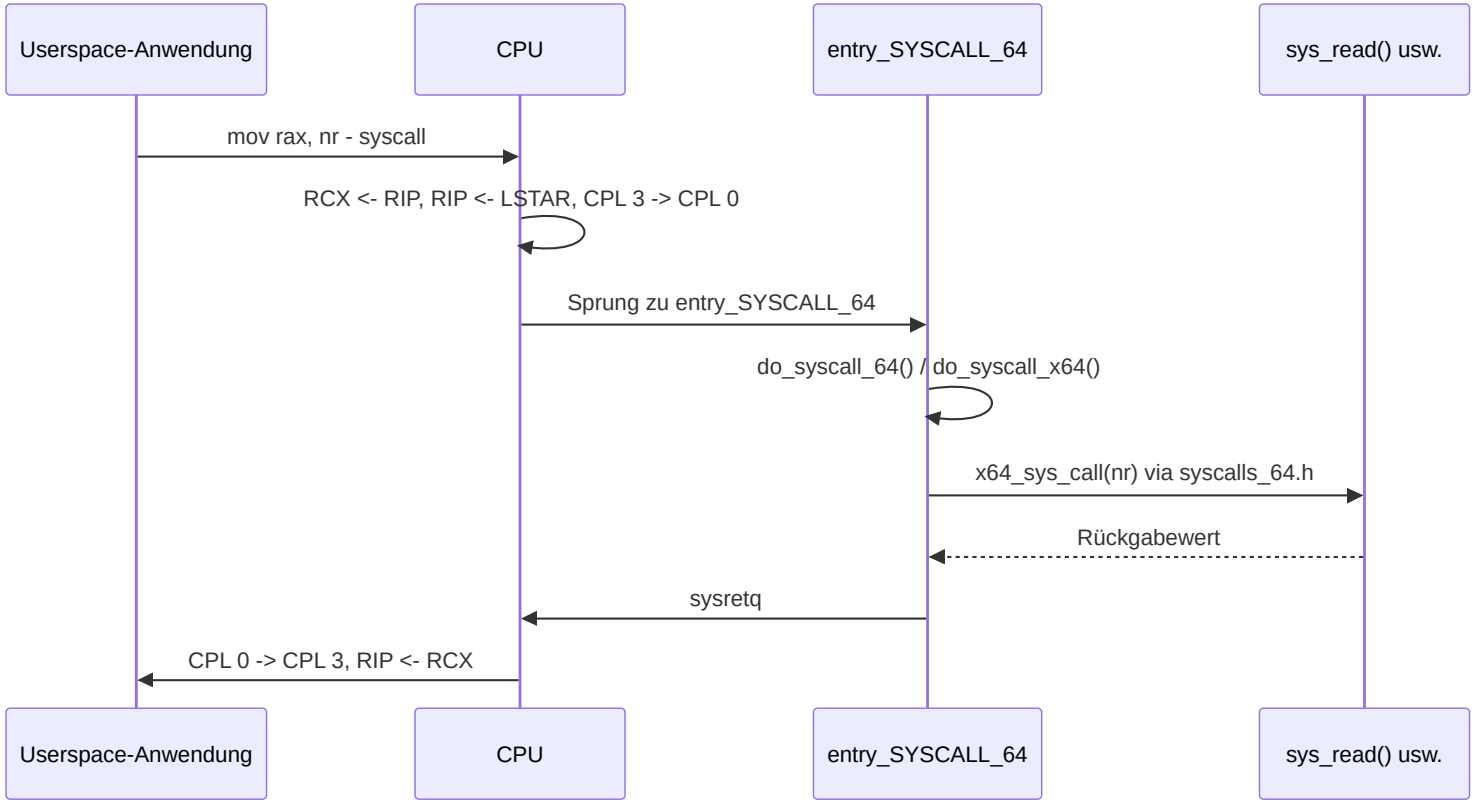
```
    call    do_syscall_64       /* returns with IRQs disabled */

...

syscall_return_via_sysret:
    ...
    SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi

    popq    %rdi
    popq    %rsp
SYM_INNER_LABEL(entry_SYSRETQ_unsafe_stack, SYM_L_GLOBAL)
    ANNOTATE_NOENDBR
    swapgs
    CLEAR_CPU_BUFFERS
    sysretq
```

Gesamtablauf: Syscall Control Flow



Linux-Kernel: Software Contract (1/2)

System V Application Binary Interface AMD64 Architecture Processor Supplement https://gitlab.com/x86-psABIs/x86-64-ABI Appendix A Linux Conventions – Appendix A.2 AMD64 Linux Kernel Conventions

1. User-level applications use as integer registers for passing the sequence RDI, RSI, RDX, RCX, r8 and r9. The kernel interface uses RDI, RSI, RDX, r10, r8 and r9.
2. A system-call is done via the `syscall` instruction. The kernel clobbers registers RCX and r11 but preserves all other registers except RAX.
3. The number of the syscall has to be passed in register RAX.
4. System-calls are limited to six arguments, no argument is passed directly on the stack.
5. Returning from the `syscall`, register RAX contains the result of the system-call. A value in the range between -4095 and -1 indicates an error, it is `-errno`.
6. Only values of class INTEGER or class MEMORY are passed to the kernel.

Linux-Kernel: Software Contract (2/2)

Siehe auch `man 2 syscall`:

The second table shows the registers used to pass the system call arguments.

ARCH/ABI	ARG1	ARG2	ARG3	ARG4	ARG5	ARG6	ARG7	NOTES
…	…	…	…	…	…	…	…	…
x86-64	rdi	rsi	rdx	r10	r8	r9	-	
x32	rdi	rsi	rdx	r10	r8	r9	-	

Beispielcode in Userspace-Anwendung (direkt) (1/2)

```
$ r2 /bin/ld.so
...
[0x00019b6b]> s 0x00019b8e

[0x00019b8e]> pd 5
            0x00019b8e      488d7c2448      lea rdi, [rsp + 0x48]
            0x00019b93      488d74244c      lea rsi, [rsp + 0x4c]
            0x00019b98      b835010000      mov eax, 0x135
            0x00019b9d      0f05            syscall
            0x00019b9f      85c0            test eax, eax

[0x00019b8e]>
```

Beispielcode in Userspace-Anwendung (direkt) (2/2)

`arch/x86/entry/syscalls/syscall_64.tbl`

```
309 common  getcpu          sys_getcpu
```

`man 2 getcpu`

```
       getcpu - determine CPU and NUMA node on which the calling thread is running

...
       int getcpu(unsigned int *_Nullable cpu, unsigned int *_Nullable node);
```

Beispielcode in Userspace-Anwendung (glibc) (1/3)

- Selten direkte Syscalls
- Typisch: über Wrapper der C-Standardbibliothek GNU libc

```
$ readelf --dyn-syms --wide /bin/ls

Symbol table '.dynsym' contains 138 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
   ...
    17: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND strcpy@GLIBC_2.2.5 (3)
    18: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __fpending@GLIBC_2.2.5 (3)
    19: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND flistxattr@GLIBC_2.3 (2)
   ...
```

Beispielcode in Userspace-Anwendung (glibc) (2/3)

```
$ r2 /bin/ls
...
[0x00006760]> aaa
...

[0x00006760]> axt @ sym.imp.flistxattr
fcn.000100a0 0x1038b [CALL:--x] call sym.imp.flistxattr
fcn.000100a0 0x103b3 [CALL:--x] call sym.imp.flistxattr

[0x00006760]> pd 3 @ 0x00010382
            0x00010382      660f1f440000    nop word [rax + rax]
|           ; CODE XREF from fcn.000100a0 @ 0x1011a(x)
|           0x00010388      4489e7          mov edi, r12d
|           0x0001038b      e8a03dffff      call sym.imp.flistxattr
```

Beispielcode in Userspace-Anwendung (glibc) (3/3)

- Für manche Syscalls existiert kein glibc-Wrapper
- Beispiel: `futex`

```
NAME
       futex - fast user-space locking

...

SYNOPSIS
       ...
       long syscall(SYS_futex, uint32_t *uaddr, int futex_op, uint32_t val,
                    const struct timespec *timeout,   /* or: uint32_t val2 */
                    uint32_t *uaddr2, uint32_t val3);

       Note: glibc provides no wrapper for futex(), necessitating the use of syscall(2).
```

ANALYSEMÖGLICHKEITEN

Zwei Perspektiven

STATISCH (DISASSEMBLIEREN)	DYNAMISCH (EBPF)
Was *könnte* ein Programm potenziell aufrufen?	Was wird *tatsächlich* ausgeführt?
Binary auf der Festplatte	Laufendes System über Zeit
(Grobe) Obergrenze	Reale gemessene Häufigkeit

Statische Analyse (Disassemblieren)

Ansatz: Syscalls aus Binaries lesen

- Alle ELF-Binaries des Systems disassemblieren
- In den ausführbaren (`.rx`)-Sektionen nach `syscall` -Instruktionen suchen
- Vorausgehenden `mov rax, <n>` als Syscall-Nummer auflösen
- Daten in sqlite speichern

Was eine statische Analyse liefert

- **Grobe Obergrenze**: Welche Syscalls sind überhaupt im Code vorhanden?

Grenzen der statischen Analyse: Ungenauigkeiten

- Inkorrektes Disassemblieren, IO unvorhersehbar
- Zur Laufzeit berechnete Syscall-Nummern (`rax` nicht konstant)
- JITs / dynamisch generierter / selbstmodifizierender Code
- Interpretierte Sprachen (Java, Python) – Syscalls sind in der Runtime
- Allgemein: Syscalls oft in dynamisch gelinkten Bibliotheken (Glibc)
- vDSO (z. B. `clock_gettime` , `gettimeofday`): läuft rein im Userspace ohne Kernel-Trap – weder hier noch bei dynamischer Analyse sichtbar
- Hier analysiert: nur direkte Syscalls im Programm
  - Keine Control-Flow-Analyse
  - Keine Abschätzung, ob ein Programm Bibliotheken lädt, die Syscalls aufrufen

Dynamische Analyse mit eBPF

Was ist eBPF?

Gute Quelle: https://ebpf.io/what-is-ebpf/ oder https://cfp.tuebix.org/tuebix-2026/talk/CZERNF/

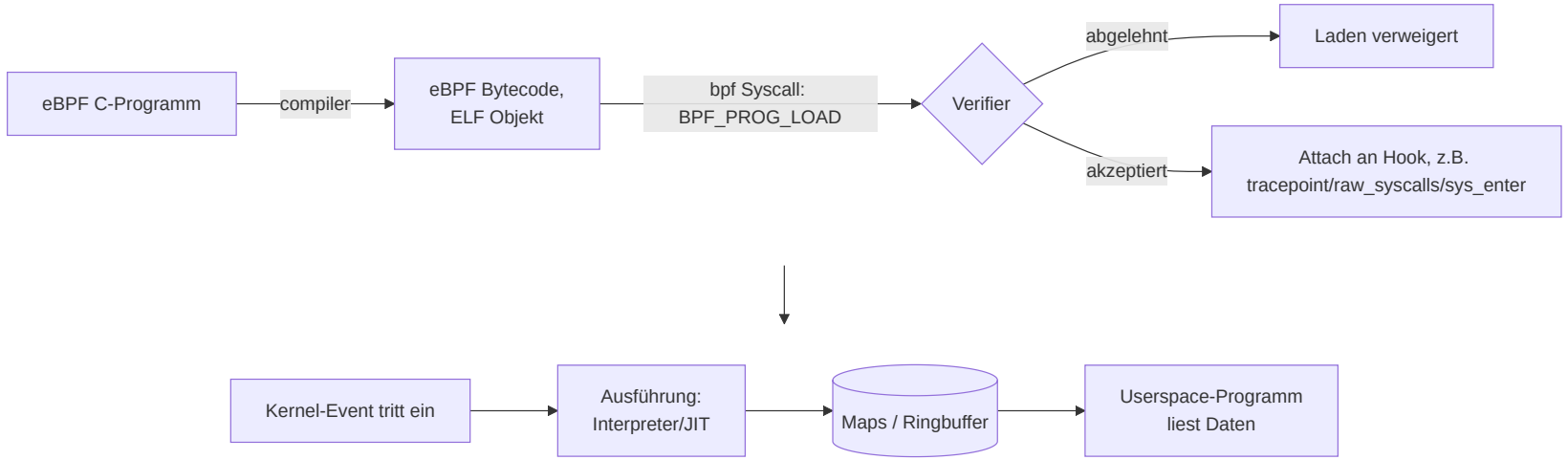
- Extended Berkeley Packet Filter
- Sandboxed Programme, die im Kernel laufen
- Kompiliert zu eBPF Byte Code
- Verifier prüft vor dem Laden: kein Crash, keine Endlosschleife
- Durch verschiedene Hooks/Events getriggert

  z. B. vordefinierte Tracepoints wie hier `tracepoint/raw_syscalls/sys_enter`
- Use Cases: z. B. Networking / Packet Filtering, System Introspection, Tracing, Performance Profiling
- Linux-Kernel-Dokumentation in `Documentation/bpf/`

eBPF-Ausführung im Kernel

- Interpretiert (Virtuelle Maschine) oder compiled (JIT)
- Verifier: `kernel/bpf/verifier.c`
- Loader: `bpf` syscall
- Interpreter/VM: `kernel/bpf/core.c` `___bpf_prog_run()`
- JIT Compiler: architekturspezifisch; z. B. `arch/x86/net/bpf_jit_comp.c` `bpf_int_jit_compile()`

eBPF: vom Quellcode zur Ausführung



eBPF-Nutzung

Unter anderem:

- libbpf: Referenzbibliothek für eBPF Development; C
  https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/lib/bpf?h=v7.1
  https://github.com/libbpf/libbpf
- bcc: Tools für "IO analysis, networking, monitoring, and more"
  https://github.com/iovisor/bcc
- bpftrace: eigene DSL
  https://github.com/bpftrace/bpftrace

eBPF-Nutzung: Beispiele

BCC:

```
sudo apt install bpfcc-tools
sudo execsnoop-bpfcc
sudo syscount-bpfcc -i 1
```

bpftrace: https://bpftrace.org/tutorial-one-liners#lesson-3-file-opens

```
bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm, str(args.filename)); }'
```

eBPF ELF Binary (1/2)

Beispiel anhand der eBPF-Komponente vom Programm zur dynamischen Analyse: Disassemblieren mit Objdump:

```
$ llvm-objdump -d syscalls_dynamic.bpf.o
...
syscalls_dynamic.bpf.o: file format elf64-bpf
...

Disassembly of section tracepoint/raw_syscalls/sys_enter:

0000000000000000 <handle_sys_enter>:
       0:       79 13 08 00 00 00 00 00 r3 = *(u64 *)(r1 + 0x8)
       1:       b7 02 00 00 00 00 00 00 r2 = 0x0
       ...

Disassembly of section tracepoint/sched/sched_process_exec:

0000000000000000 <handle_exec>:
       0:       bf 16 00 00 00 00 00 00 r6 = r1
       1:       85 00 00 00 0e 00 00 00 call 0xe
       ...
```

eBPF ELF Binary (2/2)

Beispiel anhand der eBPF-Komponente vom Programm zur dynamischen Analyse: Disassemblieren mit radare2:

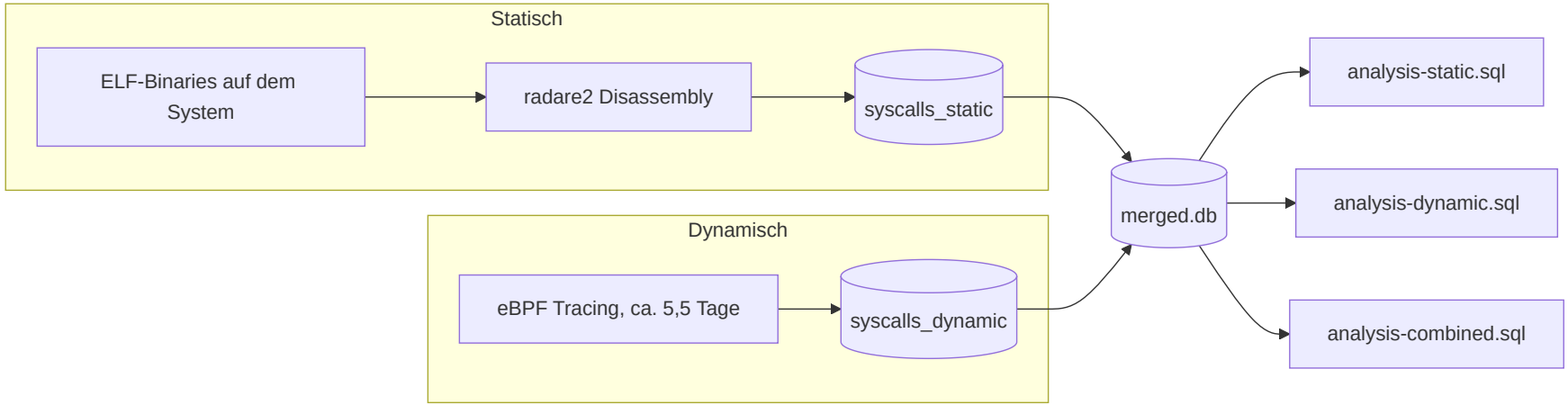
```
$ r2 syscalls_dynamic.bpf.o
...
[0x08000040]> aaa
...
[0x08000040]> i
...
arch     bpf
...
machine  Berkeley Packet Filter

[0x08000040]> pd
            ;-- section..text:
            ;-- section.tracepoint_raw_syscalls_sys_enter:
            ;-- pc:
┌ 400: sym.handle_sys_enter ();
; [03] -r-x section size 400 named tracepoint/raw_syscalls/sys_enter
│           0x08000040      7913080000..   ldxdw r3, [r1+0x8]          ; syscalls_dynamic.bpf.c:84:20    key.syscall_nr
│           0x08000048      b702000000..   mov64 r2, 0x0               ; syscalls_dynamic.bpf.c:40:54    struct task_str
```

DAS EXPERIMENT

Ansatz

- Statische Analyse: Disassemblieren jeder ELF-Binary auf dem System
- Dynamische Analyse: Syscall-Tracing mit eBPF für ein paar Tage (ca. 5,5)
- Beides in sqlite abspeichern, und dann analysieren



Systeminformationen

Setup basierend auf Debian 13; einige installierte Pakete:

- Build-essential
- Programmiersprachen: Rustup, Go, Clang, Zig, Java, Python, …
- Browser: Chromium, Firefox
- Hypervisor
- VMs (Linux/Windows)
- Viele weitere Programme

$\Rightarrow$ viel, aber nicht unbedingt repräsentatives / typisches Verhalten

Quellcode

Disclaimer: hauptsächlich durch KI erstellt

```
├── analysis
│   ├── analysis-combined.sql
│   ├── analysis-dynamic.sql
│   └── analysis-static.sql
├── CLAUDE.md
├── dbs
│   ├── merged.db
│   ├── syscalls-dynamic-....db
│   ├── syscalls-dynamic-....db
│   ├── syscalls-dynamic-....db
│   └── syscalls-static.db
├── merge-dbs
│   ├── ...
├── syscalls-dynamic-analysis
│   ├── ...
└── syscalls-static-analysis
    ├── ...
```

Statische Analyse

Python-Skript

- Sucht alle ELF Binaries auf dem System
- Analysiert diese mit radare2 (capstone Disassembler)
- Extrahiert syscalls mit radare2
- Unterstützt unterschiedliche Syscall-Aufrufarten (64/32 bit, interrupt/sysenter/syscall, …)

Dynamische Analyse

C-Programm

- eBPF CO-RE
- Tracepoint `raw_syscalls:sys_enter`
- Zusätzlich Systemd-Service zur frühen Ausführung beim Boot

AUSWERTUNG

Datenbanken

Grobe Größe der Dateien:

```
$ du -h dbs/syscalls-static.db
2,4M    dbs/syscalls-static.db

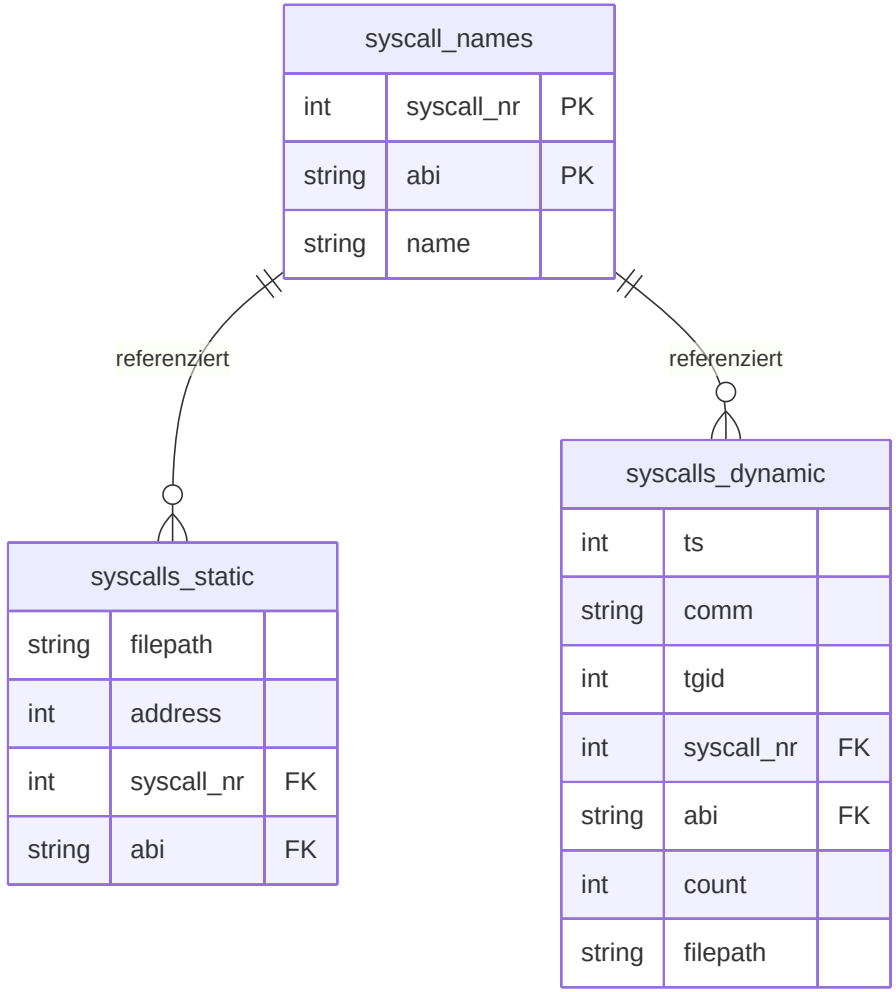
$ du -ch dbs/syscalls-dynamic*.db | tail -n1
590M    insgesamt

$ du -h dbs/merged.db
1,1G    dbs/merged.db
```

Datenbankschema

```
$ sqlite3 dbs/merged.db .schema
```

```
CREATE TABLE syscall_names    ( ... );
CREATE TABLE syscalls_static  ( ... );
CREATE TABLE syscalls_dynamic ( ... );
CREATE INDEX ix_syscalls_dynamic_nr_abi
    ON syscalls_dynamic(syscall_nr, abi, count);
CREATE INDEX ix_syscalls_dynamic_filepath_comm
    ON syscalls_dynamic(filepath, comm, syscall_nr, count);
```


syscall_names
int
syscall_nr
PK
string
abi
PK
string
name
referenziert
referenziert
syscalls_static
string
filepath
int
address
int
syscall_nr
FK
string
abi
FK
syscalls_dynamic
int
ts
string
comm
int
tgid
int
syscall_nr
FK
string
abi
FK
int
count
string
filepath

Datenanalyse mit SQL

```
$ sqlitebrowser --read-only dbs/merged.db
```

```
├── analysis-static.sql
├── analysis-dynamic.sql
└── analysis-combined.sql
```

FAZIT (1/2)

- Theoretische Hintergründe und praktisches Experiment, um Interesse zu wecken
- Statische Analyse:
  - Typischerweise über Bibliotheken/Runtimes/Interpreter (glibc, Go-Runtime, …)
  - Fehler beim Disassemblieren
  - False Positives: Daten werden als Code interpretiert
  - False Negatives: "Indirekte" Syscalls / nicht offensichtlicher Kontrollfluss
- Dynamische Analyse: Hier am häufigsten z. B.
  - Concurrent Programming: Mutex (`futex`), `sched_yield`
  - IPC: `sendmsg` / `recvmsg`
  - Device Control: `ioctl`
  - Generische Syscalls wie `write` / `read`

FAZIT (2/2)

Beide Perspektiven zusammen ergeben ein genaueres Gesamtbild

Fragen/Anmerkungen/Kontakt?: stefan.walter@syss.de

THE PENTEST EXPERTS

WWW.SYSS.DE