

ubsan, kasan, syzkaller und co

Florian Westphal

Mai 2016

```
4096R/AD5FF600 fw@strlen.de  
1C81 1AD5 EA8F 3047 7555  
E8EE 5E2F DA6C F260 502D
```

1. kernel debug optionen
2. Tests via compiler-Erweiterungen: ubsan, kasan, ...
3. kernel-fuzzing

SLUB Debugging

- ▶ Standard-Allokator seit 2.6.23 (Oktober 2007)
- ▶ viele Debug-Optionen, aktivieren mit `slub_debug` boot-parameter
 - ▶ Z: Redzoning
 - ▶ P: Poisoning
- ▶ `/proc/slabinfo`, `linux/tools/vm/slabinfo.c`
- ▶ Einige Optionen auch zur Laufzeit:
`echo 1 > /sys/kernel/slab/$name/trace`
- ▶ Nur bedingt fuer Produktivsysteme geeignet (`dmesg spew`)

Fault-Injection Framework

CONFIG_FAULT_INJECT=y

- ▶ FAILSLAB, FAIL_MAKE_REQUEST, ...

- ▶ Konfiguration via debugfs:

```
ls /sys/kernel/debug/fail*  
/sys/kernel/debug/failslab:  
cache-filter ignore-gfp-wait interval probability  
space task-filter times verbose
```

- ▶ Auf einzelne Prozesse beschränkt:

```
echo 1 > /proc/pid/make-it-fail
```

- ▶ Boot-Option

Notifier Error Injection

CONFIG_NOTIFIER_ERROR_INJECT=y

- ▶ Notifier: In-Kernel Benachrichtigungen über Ereignisse
- ▶ Kernel-Module können callbacks für bestimmte Events registrieren
 - ▶ CPU x geht offline
 - ▶ neues Netzwerk-Interface wird hinzugefügt/entfernt
 - ▶ MTU-Änderung, Änderung von Features (ethtool -K)
- ▶ Notifier-Vetos möglich – "return NOTIFY_BAD"

Sonstige compile-time Kernel Debug-Optionen

- ▶ atomic-sleep
- ▶ lockdep
- ▶ soft/hard lockup Erkennung (nmi watchdog)
- ▶ linked-list debugging

AddressSanitizer

- ▶ Erkennt Fehler bei Behandlung von Speicher(addressen)
- ▶ seit gcc 4.8

```
int main(void) {  
    char xx[100];  
    xx[100] = 0;  
    return 0;  
}
```

```
gcc -fsanitize=address x.c && ./a.out
```

```
ERROR: AddressSanitizer: stack-buffer-overflow
```

```
WRITE of size 1 at ...
```

```
#0 0x40090d in main (/tmp/a.out+0x40090d)
```

```
#1 0x7fc6c0db162f in __libc_start_main
```

```
Address 0x7ffcd7a08d24 is located in stack of thread ...
```

```
Mit neuerem gcc: ASAN_OPTIONS='help=1' ./a.out
```

KernelAddressSanitizer

- ▶ `CONFIG_KASAN=y`
 - ▶ use-after-free
 - ▶ out-of-bounds Zugriffe
- ▶ $\frac{1}{8}$ Kernel RAM für Shadowing nötig
- ▶ Shadowbyte 0 → Zugeordnetes Wort allokiert
- ▶ Shadowbyte 1 → nur 1 Byte (n : nur n bytes)
- ▶ Shadowbyte < 0 : gesperrt (Free'd, Redzone, etc)
- ▶ Compiler fügt `__asan_load/_store` calls bei ALLEN Speicherzugriffen ein

KernelAddressSanitizer (cont.)

- ▶ performanter als kmemcheck
- ▶ kein erkennen v. uninitialized reads
- ▶ langsamer als SLUB debugging, findet aber mehr Fehler
 - ▶ OOB-reads
 - ▶ OOB-write: sofort, nicht erst bei freeing

Undefined Behaviour Sanitizer (UbSan)

gcc: -fsanitize=undefined, u.a.:

- ▶ signed-integer-overflow
- ▶ enum (Laufzeitfehler, wenn in einem enum ein ungültiger Wert auftaucht)
- ▶ shifts ('1<<32'), etc.

```
int main(void) { return INT_MIN / -1; }  
gcc -fsanitize=undefined x.c && ./a.out  
x.c:9:17: runtime error: division of -2147483648 by -1  
cannot be represented in type 'int'
```

Fuzzer: Trinity, syzkaller

Allg+Primitiv: Ein Programm bekommt zufällige Daten vorgeworfen

- ▶ Kernel hat mehrere Stellen an denen Daten verarbeitet werden
 - ▶ Syscalls
 - ▶ Syscall-Daten: ioctl, setsockopt
 - ▶ Inhalte von buffern (netlink, raw sockets ...)
 - ▶ Netzwerktraffic, Speichermedien, ...

```
syscall(random(), random(), random(), random());
```

... funktioniert eher nicht

syscall-Fuzzer, zufällige syscalls mit zufälligen argumenten, aber ...

- ▶ nicht alles ist zufällig
- ▶ erstellt diverse File-Deskriptoren (files, popes, sockets, etc)
- ▶ Liste mit syscalls und den erwarteten Argumenten (z.B. `accept(fd, ..)`)
- ▶ Liste mit flags für syscalls (z.B. `send`, `sendto`, ...)
- ▶ permutieren der Argumente, Abfolge, ...

Problem: Kein "Einblick" über die Code-Abdeckung

afl

american fuzzy lop - compile-time instrumentation fuzzer

- ▶ `afl-{clang,gcc,g++}`
- ▶ Valide Eingabe(n) nötig (nach `afl/in`)
- ▶ `afl-fuzz -i afl/in -o afl/out programm [args]`
- ▶ Annahme: liest von `stdin`, wenn nicht: `"-f bla"`

syzkaller

Neuer syscall-Fuzzer, Prinzip ähnlich afl
<https://github.com/google/syzkaller>

- ▶ verteilt, fuzzing läuft in Qemu-VMs
- ▶ syscalls und Argumente werden beschrieben
- ▶ (per)mutation von generierten Programmen

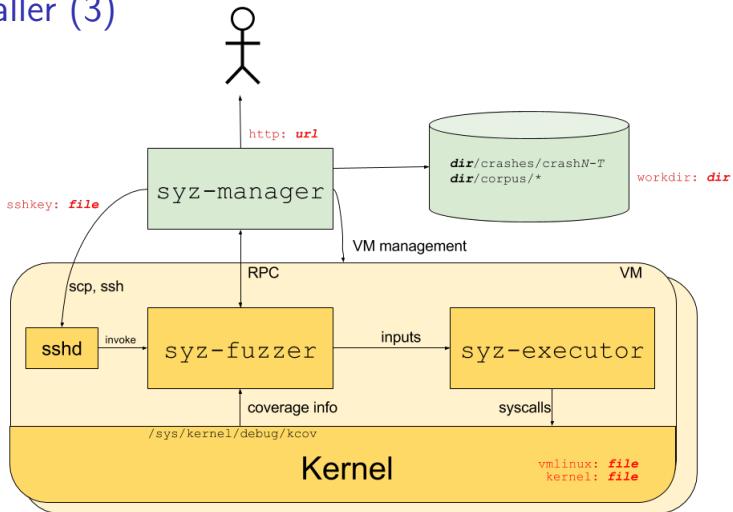
```
open(filename, f flags[open_flags], m flags[open_mode])
close(fd fd)
writev(fd fd, vec ptr[in, array[iovec_in]], vlen len[vec])
open_flags = O_RDONLY, O_WRONLY,
iovec_in {
    addr    buffer[in]
    len     len[addr, intptr]
}
```

syzkaller (2)

Voraussetzungen

- ▶ go und C++ compiler
- ▶ C-Compiler mit coverage support (gcc 6.1)
- ▶ Linux Kernel ab 4.6
- ▶ QEMU + disk image einer Distribution (vgl. `create-image.sh`)
- ▶ syzkaller tools
 - ▶ `syz-manager`: start/stop/monitoring von VM instanzen
 - ▶ `syz-fuzzer`: generiert Tests, Mutation, schickt eingaben mit mehr Coverage zum manager
 - ▶ `syz-executor`: führt Eingaben aus (d.h. eine Folge von syscalls)

syzkaller (3)



(von <https://github.com/google/syzkaller>)

syzkaller (4)

config-file (Bsp):

```
"http": "localhost:56741",
"workdir": "/home/me/syzkaller/workdir",
"kernel": "/home/me/build/linux/arch/x86/boot/bzImage",
"vmlinux": "/home/me/build/linux/vmlinux",
"image": "/home/me/syzkaller/debian.img",
"sshkey": "/home/me/syzkaller/ssh/id_rsa",
"syzkaller": "/home/me/syzkaller",
"leak": false,
"cmdline": "root=/dev/sda ro console=ttyS0",
"count": 2,
"cpu": 2,
"mem": 2048,
"enable_syscalls": [ .. ]
```

syzkaller (5)

workdir

- ▶ wird automatisch erzeugt
- ▶ corpus:
 - ▶ Eingaben die neue Ausgaben erzeugt haben
 - ▶ bleibt erhalten (beschleunigt neue starts)
- ▶ crashes:
 - ▶ dmesg oopses/warnings
 - ▶ lockups

syzkaller (6)

corpus/\$sha: autogenerierte Programme

```
mmap(&(0x7f0000000000)=nil, (0xc000), 0x3, 0x32, 0xffffffff)
clock_gettime(0x7, &(0x7f0000003000-0x10)={0x0, 0x0})
mmap(&(0x7f0000002000)=nil, (0x1000), 0x3, 0x32, 0xffffffff)
clock_gettime(0x7, &(0x7f0000003000)={0x0, 0x0})
setsockopt(0xffffffffffffffff, 0x0, 0x40, &(0x7f0000004000-
clock_gettime(0x5, &(0x7f0000005000)={0x0, 0x0})
clock_gettime(0x7, &(0x7f0000008000-0xd)={0x0, 0x0})
socket(0x2, 0x3, 0xff)
setsockopt(0xffffffffffffffff, 0x0, 0x81, &(0x7f000000c000-
```

Wenn man .c will: syz-prog2c

syskaller (7)

1. `bin/syz-manager -config ~/syzkaller/syzkaller.cfg`
2. warten – wichtige events auf stdout

```
2016/05/04 15:41:10 loaded 3225 programs
2016/05/04 15:41:10 serving http on http://localhost:56741
2016/05/04 15:41:10 serving rpc on tcp://127.0.0.1:41620
2016/05/04 15:44:40 qemu-1: saving crash 'UBSAN: Undefined
2016/05/04 15:49:18 qemu-2: saving crash 'UBSAN: Undefined
2016/05/04 15:50:56 qemu-1: saving crash 'WARNING: CPU: 3
2016/05/04 15:51:04 qemu-0: saving crash 'UBSAN: Undefined
2016/05/04 15:52:00 qemu-2: saving crash 'no output' to cra
```

Crashes/Traces unter `workdir/crashes/`

kcov: code coverage for fuzzing

`CONFIG_KCOV=y` (linux.git) gcc flag
`-fsanitize-coverage=trace-pc`

- ▶ wie auch schon bei Kasan/Ubsan: Compiler-basiert
- ▶ Kernel implementiert Funktion die dann pro block aufgerufen wird
- ▶ speichert PC in einer Art 'Log' ab
- ▶ Aufrufe aus Interrupt-Kontext werden ignoriert
- ▶ userspace muss Feature erst aktivieren → `ioctl`

kcov: code coverage for fuzzing (cont.)

```
fd = open("/sys/kernel/debug/kcov", O_RDWR);
ioctl(fd, KCOV_INIT_TRACE, COVER_SIZE);
cover = mmap(.., PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
ioctl(fd, KCOV_ENABLE, 0);
read(-1, NULL, 0); /* fuzz */
for (i = 1; i < cover[0]; i++)
    printf("0x%lx\n", cover[i]);
```

→ /sys/kernel/debug/kcov

→ addr2line

SyS_read

fs/read_write.c:562

__fdget_pos

...

KernelThreadSanitizer (KTSan)

- ▶ `fsanitize=thread`:
 - ▶ Programmausführung == Abfolge von Ereignissen
 - ▶ Speicherzugriffe
 - ▶ Synchronisierung (Locks)
 - ▶ Zustandsautomat speichert auftretende Ereignisse