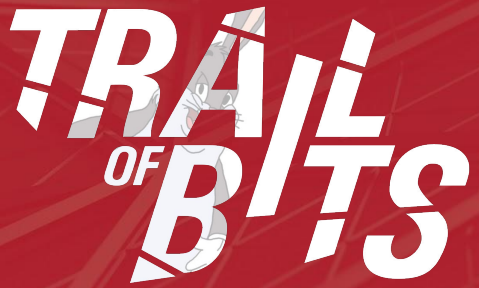


It's coming from inside the house: kernel space fault injection with KRF

Linux Security Summit 2019

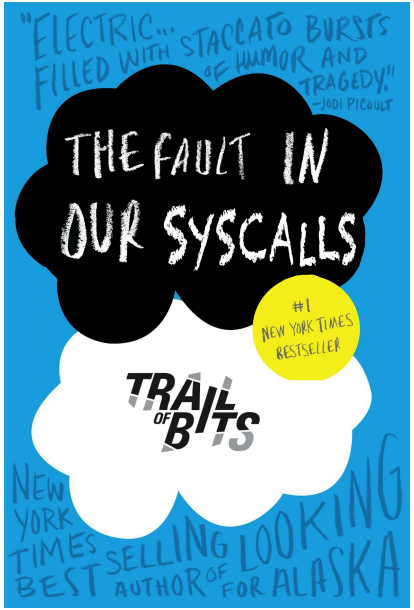
William Woodruff

Hi

- **William Woodruff (@8x5clPW2)**
  - ~~Big chungus~~ Security Engineer
  - Research & engineering at ToB
    - Research: Program analysis, automated exploitation/vulnerability reasoning
    - Engineering: Security-oriented client work, mostly open source

This talk has two parts

- **Part 1: Faults 👋 are 👋 vulnerabilities**
  - Handling faults is hard, some can't be handled
  - Failing to handle faults leads to real security vulnerabilities
  - The cloud™ makes it easier than ever to write fault-vulnerable code
    - The cloud™ itself is vulnerable to faults
- **Part 2: Doing fault injection inside the kernel with KRF**
  - How it works
  - How to use it on your tools
  - Our results



What even is a fault?

- **In the context of systems programming: well-specified failure modes for (usually) kernel managed resources**
- **Almost everything that touches the kernel can fail**
  - `open(2)`: Bad path, no more space, bad flags, interrupted by signal delivery...
  - `fork(2)`: No more memory, user/group limits, ...
  - Interactions between capabilities, different filesystems, ACLs, ...
- **Faults are part of the design contract!**
  - Hardware is fundamentally unreliable, resources are eventually exhausted, permission boundaries are eventually challenged

Handling faults is hard

- **UNIX baggage: no (real) unified fault reporting mechanism**
  - 99% of the time: clear **`errno`**, make call, check return, check **`errno`**
  - Problems:
    - Historical: **`errno`** wasn't always thread-local
    - **`errno`** doesn't always have to be cleared, so inconsistent habits formed
    - Inconsistent return values: `-1`, `NULL`, `(void *) -1`, some kind of error enum (libraries), ...
    - No enforcement: Userspace programmers get lazy and don't bother checking errors at all

Handling faults is hard, cont.

- **Some faults can't even be handled**
  - `fsync(2)`
    - "fsyncgate": https://wiki.postgresql.org/wiki/Fsync_Errors
    - Google's "solution": sidechannel the errors with a kernel module + netlink
  - `close(2)` with `EINTR`:
    - Colin Percival, 2011: "close(2) is broken"
    - 2019: Galaxy brain solution involving cookies and a pipe
  - `EINTR` + anything, really
    - signals, even signalfd
    - Usually safer to just die

Faults are an exploit primitive

- Heap spray + `read(2)` fault = arbitrary deserialization/execution:

```
{
  char *buf = malloc(4096); // sprayed buffer
  read(fd, buf, 4095);      // EWHATEVER, buf unmodified
  // ...
  yaml_parse(buf);          // arbitrary deserialization
}
```

...but faults are also rare :(

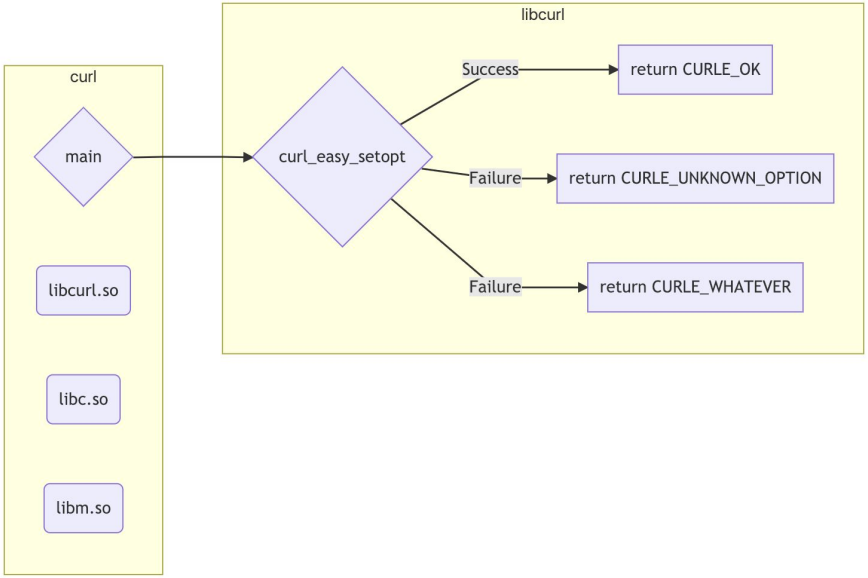
- **Normal programs perform ~thousands to ~billions of syscalls, very few fail**
  - Even fewer can be made to fail predictably
  - Even fewer fail in exploitable ways
- **...but maybe not as rare as you think**
  - Containers inherit resources (+ limits) from host system
  - Security systems (capabilities, seccomp, ...) add resource restrictions/failure modes
  - User-facing software: users do dumb things, like unplugging hot peripherals
  - *nix user model doesn't stop programs from (un)intentionally clobbering each other

Making faults less rare

- **We're interested in faults for vulnerability and resiliency research**
- **We don't want to wait for them to happen**
- **Let's do fault injection!**
- **A few potential approaches:**
  - Relink the program with faulty functions/wrappers
    - We don't always have the source :(
  - `LD_PRELOAD`
  - Dynamic instrumentation

Fault injection with LD_PRELOAD

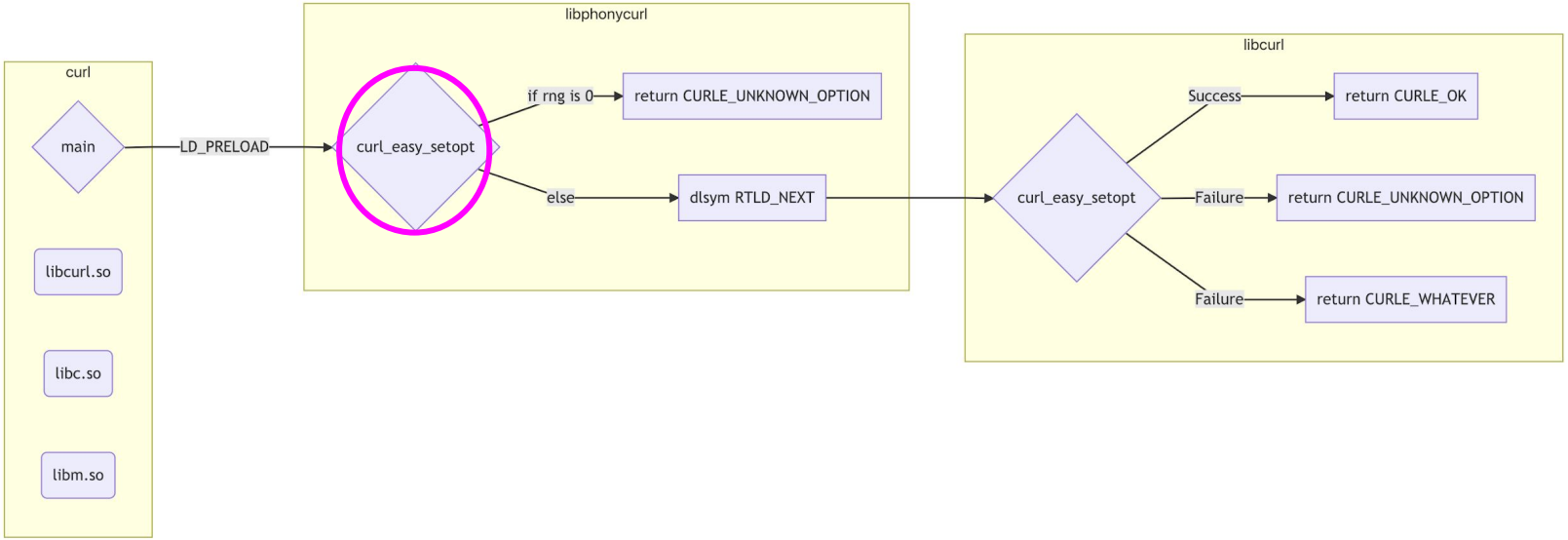
A contrived* dynamic linkage scenario



Fault injection with LD_PRELOAD

A contrived* dynamic linkage scenario, with LD_PRELOAD

```
LD_PRELOAD=./libphonycurl.so curl <whatever>
```



LD_PRELOAD: not great, not terrible

- **Pros:**
  - Conceptually simple
  - Easy to use (just a shared object)
- **Cons:**
  - Doesn't work with static binaries, and everybody loves static binaries in 2019
  - Doesn't work with `syscall(2)` or `__asm__` intrinsics
  - Unintuitive interposition: `open(3)` is `openat(2)`, `fork(3)` is `clone(2)`, ...
  - Maintaining state is a PITA, especially with MT/MP
    - `__attribute__((constructor))` is a footgun

What else is there?

- **Dynamic instrumentation**
  - `ptrace(2)` is awesome
    - ...but slow (2-3x syscall overhead best case), makes debugging hard(er)
  - Lots of dyninst frameworks (DynamoRIO, PIN, ...)
    - Performance varies, correctness varies, frameworks take a long time to learn
  - eBPF? kprobes? seccomp? Kernel debug points? LSMs?
    - Probably lots of good/fast approaches; mixed documentation and sequestered knowledge
- **Can we do better?**  **?**

Introducing KRF

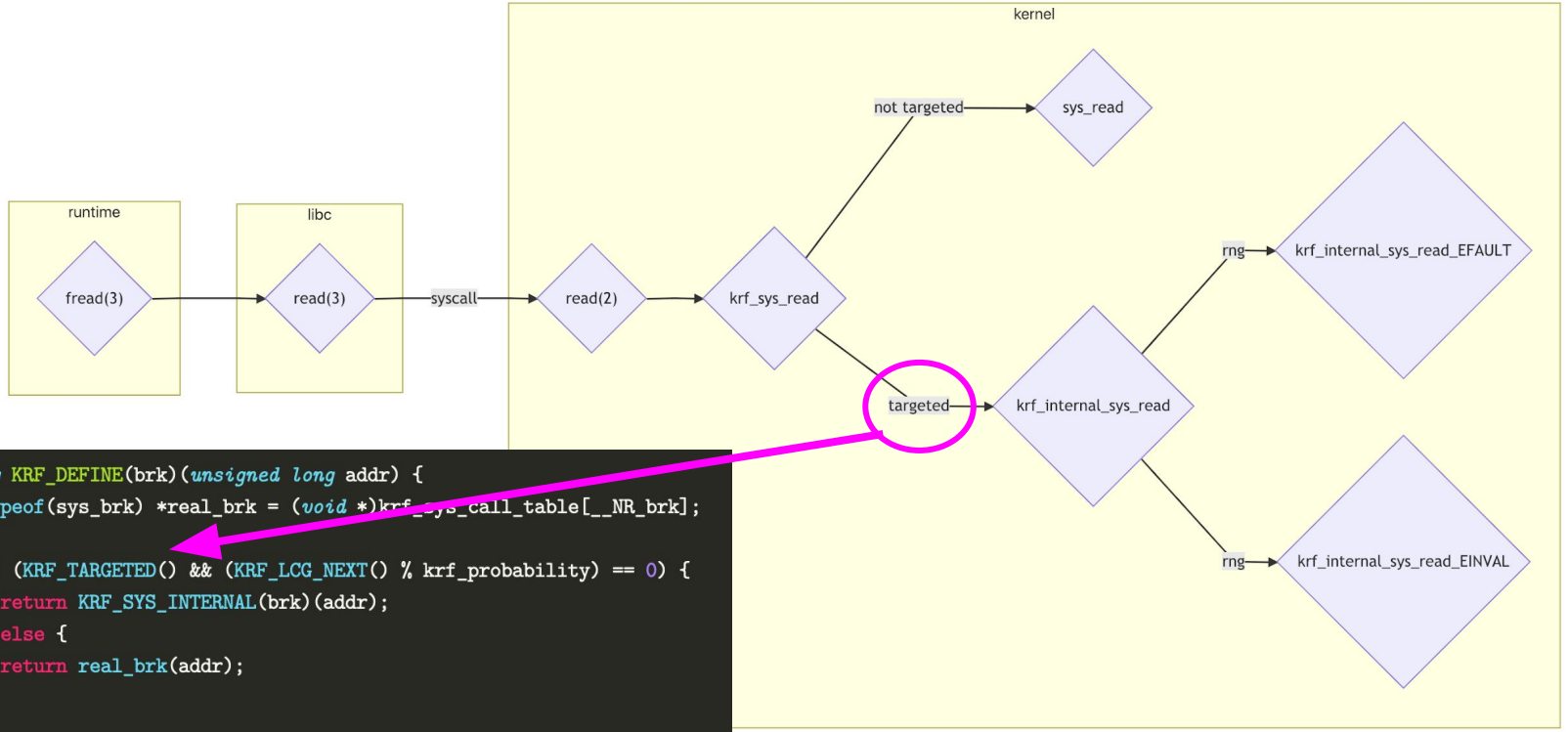
- **Kernelspace Randomized Faulter**
- **Basic process:**
  - Get the address of `sys_call_table`
  - Replace our slots of interest with wrappers
  - Wrapper: if the call is targeted, redirect to a faulty syscall that returns some `errno`
  - If the call isn't interesting, redirect to the normal syscall
  - On module unload, restore table to its original state

KRF from 1000 feet

Pseudocode:

```
asmlinkage long wrap_sys_read(...) {
  return (some_check() ? sys_read(...) : -EFAULT);
}
module_init() {
  sys_call_table = kallsyms_lookup_name(“sys_call_table”);
  sys_call_table[__NR_read] = (void*)&wrap_sys_read;
}
```

KRF's wrapping/interception mechanism




```
long KRF_DEFINE(brk)(unsigned long addr) {
  typeof(sys_brk) *real_brk = (void *)krf_sys_call_table[__NR_brk];

  if (KRF_TARGETED() && (KRF_LCG_NEXT() % krf_probability) == 0) {
    return KRF_SYS_INTERNAL(brk)(addr);
  } else {
    return real_brk(addr);
  }
}
```

KRF: Targeting strategies

- **Target a particular user/group by uid/gid**
  - Convenient `current_uid()`/`current_gid()`/etc macros
  - Extra hassle when dealing with a multi-process, multi-user application
- **Target a PID or inode**
- **Custom `personality(2)`**
  - Exists specifically to provide different syscall behavior based on process disposition
    - `PER_BSD`, `PER_SUNOS`, `PER_XENIX` (lol)
  - Children inherit personality, so simple as `personality(2)` + `exec`

Using KRF

- **Setup: for 99% of you, as simple as:**
  - `git clone https://github.com/trailofbits/krf && cd krf`
  - `make && sudo make install`
  - `sudo make insmod`
- **Three userspace components: `krfctl`, `krfexec`, `krfmesg`**
  - `krfctl`: Set module parameters
  - `krfexec`: Run an arbitrary program with KRF's telltale personality(2)
  - `krfmesg`: Read events from KRF's netlink socket

Using KRF

- `krfctl`: Choose your fighter(s)
  - `sudo krfctl -F 'read,write,open,close'`
  - `sudo krfctl -P ipc`
  - `sudo krfctl -T PID=1`
  - `sudo krfctl -c`

- `krfexec`:
  - `krfexec grep …`
  - `krfexec firefox`

Does KRF work?

- **Yes!**
  - Finds vulnerabilities in native components during smart contract audits
  - Found a DoS in Kubernetes during ToB's audit
    - Much love to Bobby Tonic for doing all the hard distributed work
- **No, but maybe soon!**
  - Trashes your programs in completely unrealistic ways
    - Good for finding bugs, bad for finding vulnerabilities
  - We've had an intern working on this, has made excellent progress on triage

Thank you!

William Woodruff

Security Engineer

github.com/woodruffw

william@trailofbits.com

References/Links

LD_PRELOAD is super fun. And easy!
https://jvns.ca/blog/2014/11/27/ld-preload-is-super-fun-and-easy/

Kernel tracing with eBPF
https://media.ccc.de/v/35c3-9532-kernel_tracing_with_ebpf

Intercepting and Emulating Linux System Calls with Ptrace
https://nullprogram.com/blog/2018/06/23/

How to write a rootkit without really trying
https://blog.trailofbits.com/2019/01/17/how-to-write-a-rootkit-without-really-trying/

SECure COMPuting with filters
https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt

References/Links

KRF
https://github.com/trailofbits/krf

Hooking the Linux System Call Table
https://tnichols.org/2015/10/19/Hooking-the-Linux-System-Call-Table/

Linux on-the-fly kernel patching without LKM
http://phrack.org/issues/58/7.html

close(2) is broken
https://www.daemonology.net/blog/2011-12-17-POSIX-close-is-broken.html

fsyncgate: errors on fsync are unrecoverable
https://danluu.com/fsyncgate/