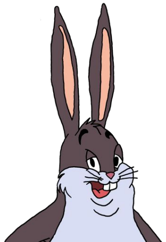
Going sicko mode on the linux kernel

Empire Hacking 2/12/19

William Woodruff

Hi

- **William Woodruff (@8x5clPW2)**
  - Security Engineer at Trail of Bits
  - Research & Development and Engineering practices
    - R&D: Software resiliency, automated program/vulnerability analysis
    - Engineering: osquery, client contracting
  - Open source work
    - Homebrew
    - ToB: KRF, twa, WInchecksec, osquery, nginx, ...
    - Personal: lots of projects



Faults

- **Subclass of bugs**
- **Distinguishing feature: introduced by function/operation failure, rather than *direct* user input**
    - Think: `read()`, `write()`, or `malloc()` failures: user didn't do anything wrong*, the system just couldn't service the request for whatever reason
        - Part of the contract for using those functions!
    - Hardware is fundamentally unreliable, parts of the system that interact with hardware must either be resilient or report failures upwards (or both)
    - POSIX and NT both have unified failure reporting mechanisms (`errno` and `GetLastError`), but actual usage is mixed

How many potential faults?

```
int main(void) {
  chdir(getenv("TMPDIR"));
  int fd = open("hello", O_WRONLY);
  write(fd, "hello tmpdir\n", 13);
  lseek(fd, 6, SEEK_SET);
  do_more(fd);
  return 0;
}
```

???

- Each of these calls can fail!
- If just one fails, each after is likely to fail (or do the wrong thing):
  - `chdir()` fails: `open()` either fails or creates the file in the wrong place
  - `write()` fails: `lseek()` now has an invalid offset (fd unchanged)

Faults are **not** limited to elementary calls like these!

Faults as a vulnerability class?

- **Failure to handle faults leads to potential vulnerabilities:**
  - NULL dereferences: many functions return NULL on failure
  - Uninitialized memory usage: give a function a buffer, fail to check whether the function populated it
- **As a whole, not as exploitable**
  - Faults are uncommon, hard to predict

Faults as a vulnerability class?

Heap spray + faulty read + normally trustworthy call = exploit:

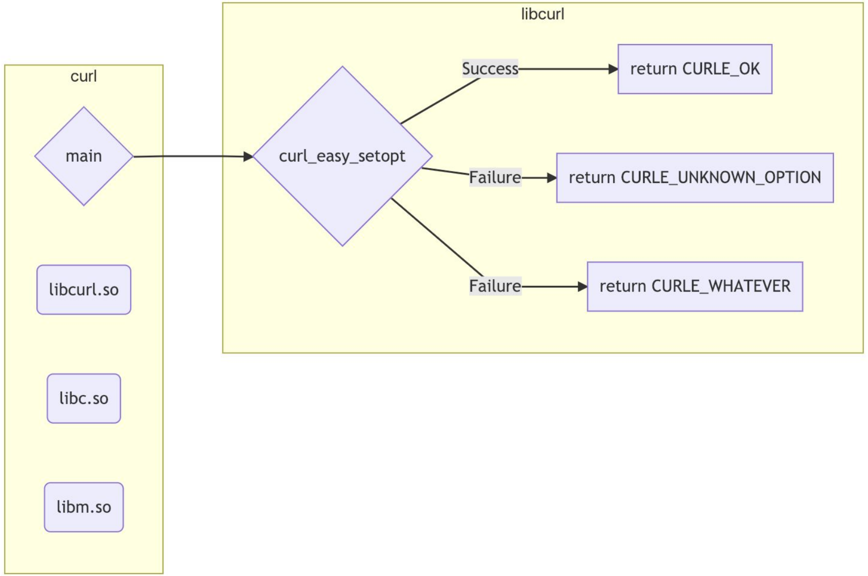
```
{
  char *buf = malloc(4096); // sprayed buffer
  read(fd, buf, 4095);      // EFOOBAR, buf unmodified
  // ...
  yaml_parse(buf);          // arbitrary deserialization
}
```

Fault injection

- **Instead of waiting for faults to happen, make them happen!**
  - Prereq: Need to know which functions we want to fault
- **Many different approaches:**
  - Relink/load the program with faulty versions of the targeted functions
  - Use `LD_PRELOAD` to dynamically load faulty functions instead
  - Dynamically instrument the program to redirect the targeted functions
    - Userspace or kernelspace (more on this one later)

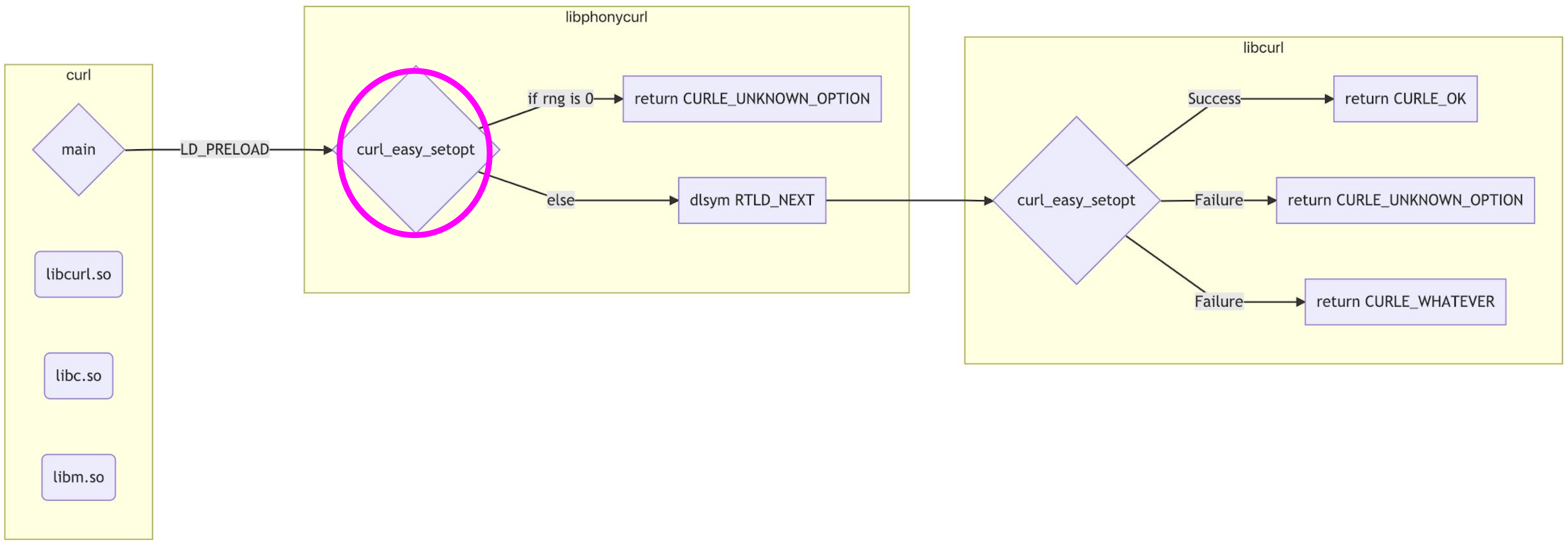
First approach: linkage fiddling

A contrived* dynamic linkage scenario



First approach: linkage fiddling

A contrived* dynamic linkage scenario, with LD_PRELOAD



Fault injection for software resiliency

- **Systems have countless failure modes**
  - Failure modes cross process, application, network, user boundaries
  - Most are not obvious until they actually happen
- **Fault injection into active systems improves system resiliency**
  - How many failures can occur before unacceptable service deterioration happens?
  - “Chaos Engineering”

Fault injection for software resiliency

- **What happens if we randomly turn off services? VMs? Entire datacenters?**
  - Chaos Monkey/Simian Army designed at Netflix for this purpose

- **Faulting just C/C++ calls doesn't catch "rich" failure modes in managed languages**
  - All languages with nullable types need additional logic to avoid exceptions/segfaults
  - JVM fault injection (nulls, random latency): https://github.com/mrwilson/byte-monkey
- **Fault the hardware: memory errors, instruction set extensions, make peripherals unreliable, ...**

Fault injection for vulnerability research

- **Can we find exploitable bugs by injecting faults into programs?**
  - Probably lower profile overall: denial, degradation of service
  - Need to act like a fuzzer (randomly induce faults to explore program space)
    - ...but also want determinism (want to reliably repro a faulty run)
- **Problem: it's easy to cause a fault, hard to make it exploitable**
  - Hard to make a local file `read()` fail from across the network
  - Hard to make `malloc()` fail on systems with overcommit*

Practical fault exploitation

- **Resource limits**
  - Capping resources is a popular way to protect against DoS from exhaustion
    - ...but once you add caps, you need to make sure every call succeeds!
- **POSIX user model weaknesses**
  - We almost never want random processes to interact, but nothing stops them
  - If we're already running as the user, we can modify (almost) all of their resources
    - Leverage TOCTOU
    - Delete/lock files while they're being read/written to
    - Spawn processes with the same name/squat on pipes and sockets
  - Escalation? Target calls that cooperate with more privileged processes

Exploitation blockers

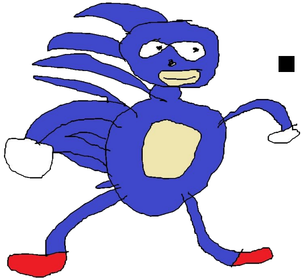
- **Wrapping/replacing calls with `LD_PRELOAD` is imperfect:**
    - `LD_PRELOAD` can't intercept statically linked calls — they're not dynamically loaded!
    - `read()` and `write()` are syscalls, but `LD_PRELOAD` can only see their libc wrappers
        - Unintuitive wrapping: glibc's `open(3)` is really `openat(2)`, `fork(3)` and `vfork(3)` are really `clone(2)`, &c.
        - Direct syscalls (e.g. `syscall(SYS_read, …)` or asm) need to be handled separately
    - Easy to make a hash of things when an `LD_PRELOAD`ed process forks (especially if you mix `LD_PRELOAD` with `__attribute__((constructor))` or equivalent
- **Other dynamic loader tricks out there, most (all?) with similar issues**
    - Solution: forget the dynamic loader, go directly to the syscalls themselves

Reliable syscall faulting

- **A few different options**
  - Source available? Statically link everything and substitute our own libc-like wrappers
    - Doesn't capture syscall(3) or asm intrinsics, but otherwise not too shabby
  - Dyninst: replace/wrap `int 80h`/`syscall` isns
    - Slow, error prone (ABI changes), need to think about implementation details
  - `ptrace(2)`
    - Some stuff with `SECCOMP_RET_TRACE`
    - Slow (2-3x syscall overhead), inferior processes only, hard to debug
  - We need to go deeper

Reliable syscall faulting

- **More options in kernelspace:**
  - kprobes can attach to syscalls, eBPF can marshal events to userspace
    - Use `bpf_probe_write_user` to rewrite return values to fake an error
    - `SECCOMP_RET_ERRNO` as well?
    - Probably pretty fast, but bcc is hard to use and kprobes docs are mixed
  - Write a rootkit
    - Rewrite the syscall table, add wrappers that invoke the faulty call when a command comes from a targeted user/group/process/whatever
    - Wicked fast, probably breaks a bunch of stuff
      - Let's do it anyways



Futzing with the syscall table

- **Basic process: get the address of `sys_call_table`, change the `__NR_<syscall>` slots we're interested in to our fake syscalls**
    - Each fake syscall just immediately faults with some valid `errno`
- **Wrappers then check to see whether we want to target a particular call and invoke the faulty version if so**
    - If not targeted, invoke the normal syscall
- **Once done (module unload), we restore the syscall table to its original state**
    - Essentially just three `memcpy()`s: save the table, clobber it, and then restore it

Futzing with the syscall table

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;
}
```

Syscall targeting options

- **Target a particular user/group by uid/gid**
  - Convenient `current_uid()`/`current_gid()`/etc macros
  - Thinking about effective users and groups makes my brain hurt + NFS insanity
  - Extra hassle when dealing with a multi-process, multi-user application
- **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`
- **Probably others, you should experiment**

Tying it all together

- **KRF is a Kernelspace Randomized Faulter**
  - Can fault 64+ individual syscalls currently (large chunks of the “core” POSIX API)
  - Faulty calls are codegen’d from YAML specs and a lot of ugly macros
  - User interfaces: `krfexec` and `krfctl`

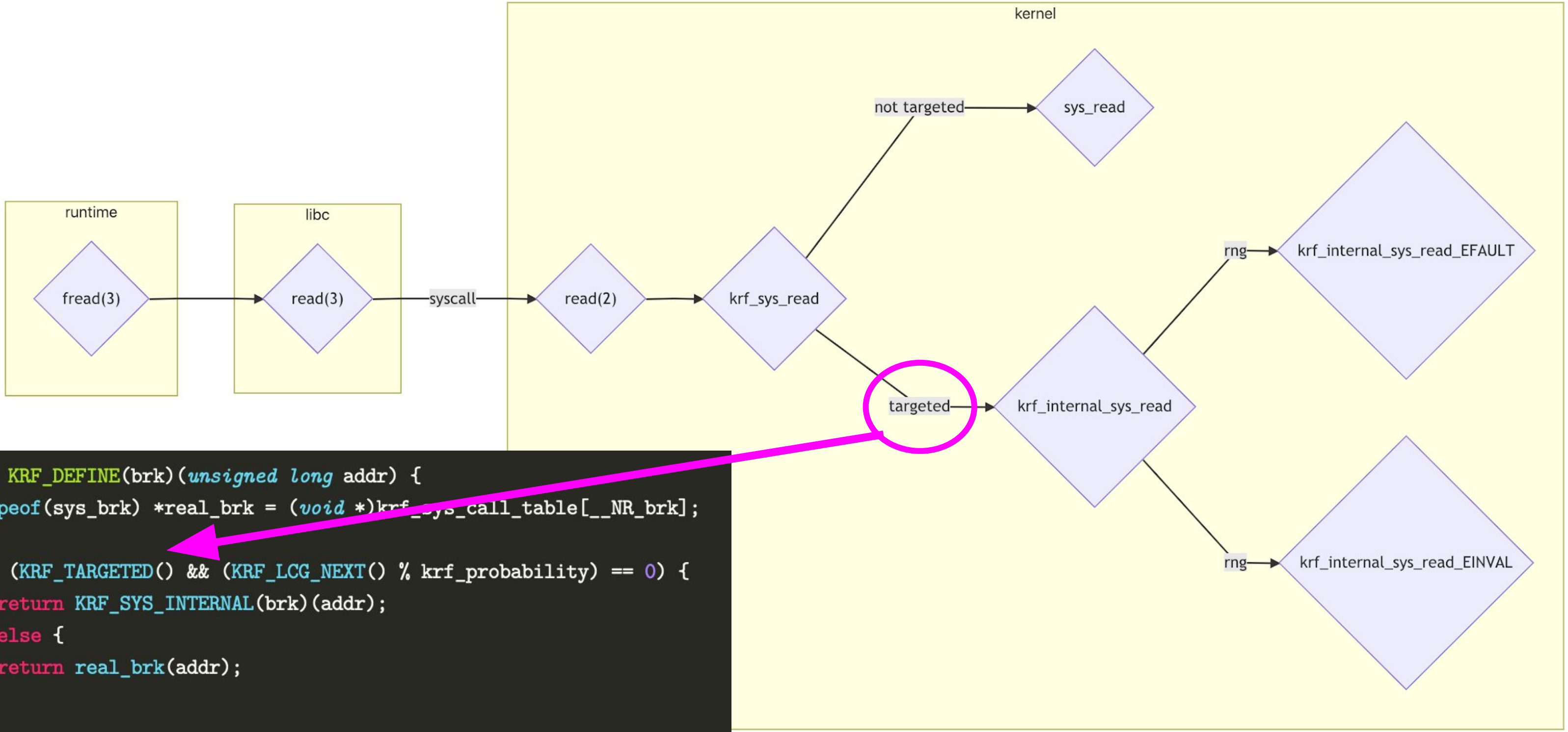
brk.yml

```yaml
proto: unsigned long addr
parms: addr
errors:
  - ENOMEM
```

```c
#define KRF_SYS_CALL brk
#define KRF_SYS_PARMS unsigned long addr
#define KRF_SYS_PARMSX addr
DEFINE_FAULT(ENOMEM) {
  return -ENOMEM;
}
static typeof(sys_brk)(*fault_table[]) = {
  FAULT(ENOMEM)
};

// Fault entrypoint.
long KRF_DEFINE_INTERNAL(brk)(KRF_SYS_PARMS) {
  return fault_table[KRF_LCG_NEXT() % NFAULTS](KRF_SYS_PARMSX);
}
```

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);
  }
}
```

krfexec and krfctl

- **krfexec: Run a program with KRF's telltale `personality(2)`**
  - `krfexec curl http://example.com`
- **krfctl: Set module parameters**
  - Which syscalls to fault
    - `sudo krfctl -F read,write,open,close # fault just these 4 syscalls`
    - `sudo krfctl -P net                   # fault all networking syscalls`
    - `sudo krfctl -c                       # clear all faulty syscalls`
  - Probability of an individual fault/RNG seed
    - `sudo krfctl -p 100      # 1/100 calls on average will fail`
    - `sudo krfctl -r 0        # set the RNG state to 0`

```
vagrant@ubuntu-bionic:/vagrant$ ./src/krfexec/krfexec htop

 Please include in your report the following backtrace:
htop(CRT_handleSIGSEGV+0x33)[0x562d4efce383]
/lib/x86_64-linux-gnu/libc.so.6(+0x3ef20)[0x7f98809f2f20]
htop(Process_display+0x1a)[0x562d4efc199a]
htop(Panel_draw+0x1ca)[0x562d4efc0dea]
htop(ScreenManager_run+0x1b5)[0x562d4efc4035]
htop(main+0x3f1)[0x562d4efbaa21]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xe7)[0x7f98809d5b97]
htop(_start+0x2a)[0x562d4efbac4a]

Additionally, in order to make the above backtrace useful,
please also run the following command to generate a disassembly of your binary:

   objdump -d `which htop` > ~/htop.objdump

and then attach the file ~/htop.objdump to your bug report.

Thank you for helping to improve htop!

Aborted (core dumped)
```

```
vagrant@ubuntu-bionic:/vagrant$ :; while [ $? -eq 0 ]; do ./src/krfexec/krfexec curl http://example.
com >/dev/null; done
  % Total    % Received % Xferd  Average SpeUnexpected error 88 on netlink descriptor 4 (address fam
ily 16)Aborted (core dumped)
```

```
vagrant@ubuntu-bionic:/vagrant$ ./src/krfexec/krfexec gcc example/chdir.c
Segmentation fault (core dumped)
```

```
vagrant@ubuntu-bionic:/vagrant$ ./src/krfexec/krfexec file ~/bash.1.gz
/etc/magic, 4: Warning: using regular magic file `/usr/share/misc/magic'
/home/vagrant/bash.1.gz: data
vagrant@ubuntu-bionic:/vagrant$ ./src/krfexec/krfexec file ~/bash.1.gz
/home/vagrant/bash.1.gz: gzip compressed data, max compression, from Unix
```

Demo

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

Contact

William Woodruff

Security Engineer

william@trailofbits.com

www.trailofbits.com

TRAIL OF BITS