CANONICAL

Securing ROS robotics platforms

March 2020

Introduction

The Robot Operating System (ROS) is a popular open-source platform for advanced robotics. Modern robots are deployed in both local warehouses and isolated remote sites similar to legacy Supervisory Control and Data Acquisition (SCADA) systems. SCADA systems are widely used in industrial (manufacturing, refining, generation, and fabrication, etc.), infrastructure (gas and oil pipelines, water treatment and distribution, etc.) and facility processes (HVAC, access, etc.). They were designed to be open, robust, easy to operate, and repair, but security was not part of the design until the proliferation of the Internet. Similar to SCADA, some of these robots are not adequately secured against attacks. By building your robot on Ubuntu, there are easy steps to secure your robot against attackers.

For this hardening exercise, we will use the TurtleBot3 platform from Robotis as the reference architecture in preparing your robot for deployment to a production environment. The focus will be on securing the underlying operating system beneath ROS 1 or 2 - Ubuntu 18.04 LTS. Although we use the Raspberry Pi based model of TurtleBot3 for demonstration purposes, most of the suggestions apply to robots based on other architectures such as x86, etc. If there are nuances related to a particular architecture, those are named explicitly in the material that follows.

Opportunistic attacks are the most prevalent types of attacks, and the majority of breaches involve attackers finding an easy target. The steps outlined herein will help you put up defences against those opportunists. Of course, there are also Advanced Persistent Threats (APTs) which are attacks that are highly customised to a particular organisation or institution. Comprehensive security against APTs is beyond the scope of this whitepaper although some items in this whitepaper should help you against APTs.

Default installation

In the instructions of the TurtleBot3, they recommend installing the Ubuntu MATE desktop. Ubuntu MATE desktop is great for a developer on their workstation. But as an operating system (OS) for a robot, a graphical user interface (GUI) is not always necessary. If possible, install Ubuntu Server 18.04 LTS. This forgoes the X11 system, which will increase the amount of available memory, reduce startup time, and reduce the attack surface due to the smaller number of packages installed. The difference between resources used and attack surface decrease, see tables 1.0-1.2 below.

	Version	Total Packages
Ubuntu Server	Ubuntu 18.04.3 LTS	496
Ubuntu MATE Desktop	Ubuntu 18.04.2 LTS*	1793

Table 1.0 - Number of packages installed
*(*Latest version available at the time of this writing)*

	Total	Used	Free	Shared	Buff/ cache	Available
Ubuntu Server	912M	106M	439M	4.4M	366M	786M
Ubuntu MATE Desktop	912M	423M	54M	9.1M	434M	415M

Table 1.1 - Memory consumption

	Size	Used	Available	Use%	Mounted on
Ubuntu Server	30G	1.5G	27G	6%	/
Ubuntu MATE Desktop	30G	4.2G	25G	15%	/

Table 1.2 - Storage consumption

By installing the Ubuntu Server image, you have fewer associated packages to update and keep secure. To do this, it is as simple as downloading the image, running dd, and booting from the SD card. Make sure to enable the SSH server. For continuing setup over ethernet, the interface is already configured for DHCP. On the other hand, if you are going to use WiFi for configuration, you need to enable it. Replace “YOUR_SSID_NAME” and “YOUR_WPA2_PASSWORD” with your values, paste the script below in your console, and hit enter to run it.

```
sudo tee /etc/netplan/01-netcfg.yaml <<EOF
network:
  version: 2
  renderer: networkd
  wifis:
    wlan0:
      dhcp4: yes
      access-points:
        “YOUR_SSID_NAME”:
          password: “YOUR_WPA2_PASSWORD”
EOF
```

Then run the following commands to bring WiFi up. Note that it will take a few seconds for the interface to associate and get an address.

```
sudo chmod 0600 /etc/netplan/01-netcfg.yaml
sudo netplan generate
sudo netplan apply
```

Remove default users such as “ubuntu”

Part of gaining access to a system remotely is attacking default usernames and simple passwords. Removing default usernames and switching to named user accounts also has the added benefit of accountability of user actions. When an account is shared it is difficult, if not impossible, to determine who exactly performed the actions that may have led to a disruption in service. It is also a good idea to install “libpam-passwdqc”, which will ensure that user passwords meet a minimum security requirement. Run the commands below to install passwdqc, create a new user, add the user to the sudo group (if required), logout as “ubuntu” user, login with the new user, and remove the default “ubuntu” user.

```
ubuntu@tb3:~$ sudo apt install libpam-passwdqc

ubuntu@tb3:~$ adduser yourNewUser
ubuntu@tb3:~$ sudo usermod -a -G sudo yourNewUser

yourNewUser@tb3:~$ sudo deluser --remove-home ubuntu
```

SSH hardening

SSH is the defacto method for connecting to a Linux server. SSH, as most know, allows for encrypted communication between client and server. However, since it does allow remote access, it also is a target for attackers. Attackers will try to perform brute force password attacks aimed at SSH connections. This is prevalent in attacks like those of the Mirai Botnet that tried a static list of 60 usernames and passwords to compromise hosts.

To prevent that type of attack, you can configure two types of mitigation; requiring ssh keys and a tool for detecting and blocking these attacks.

First, you need to generate an ssh key on your workstation and install the public ssh key on the TurtleBot3. On the workstation, run the ssh-keygen command:

```
ssh-keygen -t rsa -b 4096
```

When it asks you for the key passphrase, make sure to use a passphrase. This will encrypt the key on disk with that passphrase, which means if someone stole your private key, they would still need to know your passphrase to use it.

Next, you need to distribute the new ssh key to your TurtleBot3, and you do that via a command called “ssh-copy-id”

```
ssh-copy-id -i ~/.ssh/id_rsa.pub yourNewUser@tb3
```

Now that you have a key on TurtleBot3, you can proceed with configuring ssh daemon options in “/etc/ssh/sshd_config”. The daemon allows for extensive configuration, and to get a better understanding of the available options, run “man sshd_config”. To secure your ssh sessions, you will set the following options:

```
PermitRootLogin No
X11Forwarding no
PasswordAuthentication no
```

If you have users with no shell access, you can additionally disallow Transmission Control Protocol (TCP) and agent forwarding. Keep in mind that users with shell access can install their own forwarders.

```
AllowTcpForwarding no
AllowAgentForwarding no
```

Restart sshd

```
sudo service ssh restart
```

If you have made a mistake you may no longer be able to ssh into the TurtleBot3. In the event this happens, you can attach a keyboard and monitor to the Raspberry Pi and revert the changes made in sshd.

Next, you will want to install “sshguard” to prevent users from performing ssh brute force attacks. Why is this necessary after requiring keys? A Denial of Service (DoS) can be achieved by continual login attempts from a password attack. A tool like sshguard will block the requests to login at the firewall after several failed logins over a short period.

```
sudo apt install sshguard
```

Firewall

The ideal robotics network would be an isolated Virtual Local Area Network (VLAN) with Access Control Lists (ACL) limiting inbound and outbound traffic. This would be similar to the way SCADA environments have typically been deployed. With the truly distributed nature of robots, this is not often the case. Robots must coexist with other WiFi guests. In this case, it is necessary to apply a rule set, allowing in only ssh. To do that run “sudo ufw limit ssh”

```
jsmith@tb3:~/$ sudo ufw limit OpenSSH; sudo ufw enable
Rules updated
Rules updated (v6)
```

Note that “ufw allow ssh” would also work, but using “ufw limit”, we get an extra benefit. This way, the firewall will stall brute-force password attacks because it will start throttling new connections if it receives too many.

Home directory

The default home directory permissions on Ubuntu allow users to share files in their home directories. To prevent users from accessing other users’ files, the following changes can be made.

```
sudo chmod 0750 /home/*
sudo sed -i.orig -e ‘s/=0755/=0750/’ /etc/adduser.conf
```

Default umask

File creation and access race conditions are a way users could escalate their access beyond what they were granted. To help mitigate that, make sure to use “umask”. Users “umask” sets the file mode creation mask of processes; you can use it to restrict access to the content a given user generates. To prevent users from accessing each others files, run;

```
echo “umask 077” >> /etc/profile
```

Since not all shells interpret the “/etc/profile” file, you should also add the following line into “/etc/pam.d/login” -

```
session optional pam_umask.so umask=0077
```

While the options above set the umask for children of “bash” or PAM sessions, don’t rely on your parent process umask in your ROS code - always set your process umask explicitly (type “man 2 umask” in your console for more information on using umask in your code).

Unattended upgrades

Part of overall security hygiene is to patch security vulnerabilities in a timely manner. As ROS is based on Ubuntu, you can keep up to date with security patches by enabling unattended upgrades. All you have to do is make sure that the options below are uncommented in "/etc/apt/apt.conf.d/20auto-upgrades"

```
APT::Periodic::Update-Package-Lists "1";
APT::Periodic::Unattended-Upgrade "1";
```

This will periodically refresh the package list and upgrade packages that have security patches available. Although auto-upgrade for most packages will require no reboot, there are going to be updates that require restarts. To see if any upgrades need a reboot, you can periodically check by running the command below:

```
jsmith@dev:~$ ssh jsmith@tb3 cat /var/run/reboot-required.pkgs
linux-base
```

In this case, the "linux-base" upgrade needs a reboot; if no packages require a reboot, the file will not exist.

Disabling USB

Universal Serial Bus (USB) has been around for a long time as a convenient and quick way to expand system peripherals. This easy expansion, unfortunately, made way for USB devices that can be used for malicious intent. A couple of examples would be PoisonTap or Responder based attacks that can run on a Raspberry Pi Zero, Hack5 Turtle, or USB Armory.

One way to prevent USB abuses on your robot is to disable various USB types of devices if you are not using them. To check if you can disable individual USB types of devices, run:

```
jsmith@tb3:~$ egrep -e "USB_NET_DRIVERS=" -e "USB_STORAGE=" -e "USB_HID="
-e "USB_SERIAL=" /boot/config-`uname -r`
CONFIG_USB_NET_DRIVERS=y
CONFIG_USB_HID=y
CONFIG_USB_STORAGE=y
CONFIG_USB_SERIAL=m
```

Anything marked with a "=m" can be disabled, and anything marked with a "=y" can't because "y" means that the driver is compiled into the kernel. The example above is from a TurtleBot3 with a RaspberryPi 3+, and unfortunately, those modules need to be compiled into the kernel. The only thing we could disable is the serial devices, but the TurtleBot3 uses that module for the LiDAR communication.

We'll return to the Raspberry Pi based TurtleBot3, but if your robot is based on a different architecture, the kernel options could be different. For example, the options on the x86_64 architecture are below.

```
jsmith@x86_64:~$ egrep -e "USB_NET_DRIVERS=" -e "USB_STORAGE=" -e "USB_
HID=" -e "USB_SERIAL=" /boot/config-`uname -r`
CONFIG_USB_NET_DRIVERS=m
CONFIG_USB_HID=m
CONFIG_USB_STORAGE=m
CONFIG_USB_SERIAL=m
```

USB drivers are compiled as modules, therefore, we can disable individual types of devices. Starting with Human Interface Devices (HID), if you want to prevent someone from plugging in a keyboard or other HID devices, then block the module from loading.

```
jsmith@x86_64:~$ sudo rmmod usbhid
jsmith@x86_64:~$ echo "blacklist usbhid" |sudo tee -a /etc/modprobe.d/
blacklist.conf
```

To prevent someone from plugging in a storage device such as a USB disk or a flash drive, block off the modules from loading. Copy this script into your console and run it:

```
for i in usb-storage usb_storage; \
do sudo rmmod $i ; echo "blacklist $i" |sudo tee -a /etc/modprobe.d/
blacklist.conf;\
done
```

Disallow USB serial devices by blacklisting the USB serial driver:

```
jsmith@x86_64:~$ sudo rmmod usbserial
jsmith@x86_64:~$ echo "blacklist usbserial" |sudo tee -a /etc/modprobe.d/
blacklist.conf
```

Disallow USB networking devices by blacklisting USB networking modules:

```
find /lib/modules/`uname -r`//drivers/net/usb -type f -name \.ko | xargs
basename -s .ko | sed s'/^/blacklist /' | sudo tee -a /etc/modprobe.d/
blacklist.conf
```

Disallow all USB devices, effectively disabling USB functionality:

```
for i in $(find /lib/modules/`uname -r` -name usb -type d);\
do find $i -name *.ko | sed 's/.ko$//g' | awk -F/ '{print
"blacklist",$(NF-0)}';\
done | sudo tee -a /etc/modprobe.d/blacklist.conf
```

Now let's return to the Raspberry Pi based TurtleBot3. While you can do the above steps to disable the loading of all USB device modules, there are some limitations that you should take into consideration. If you recall the kernel configuration, some crucial modules are compiled into the kernel.

```
CONFIG_USB_NET_DRIVERS=y
CONFIG_USB_HID=y
CONFIG_USB_STORAGE=y
```

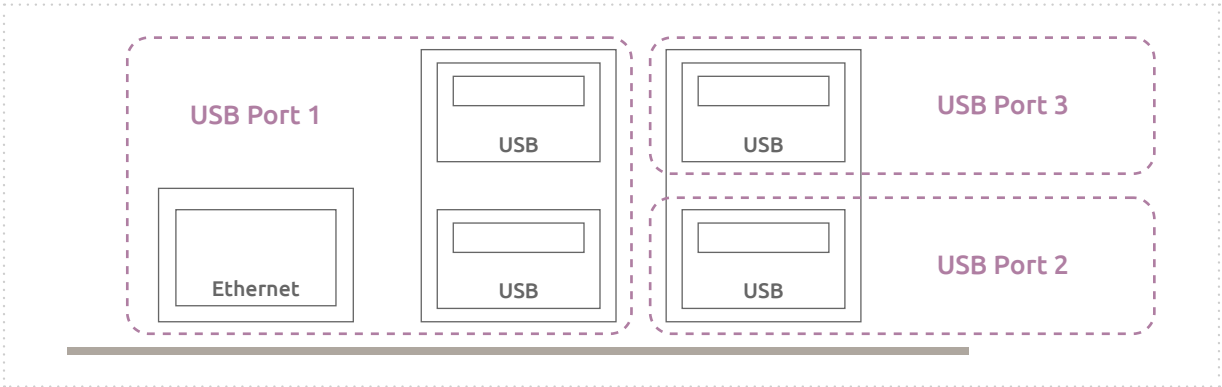
Therefore, we can't prevent someone from plugging in HID devices, and we can't preclude storage devices from being added. There is a bit more of a grey area for network devices. While the "USB_NET_DRIVERS" compiles the "usbnet" module into the kernel, we can't blacklist it. But usbnet is only a "base" driver, and a network device will require an additional chip-specific driver. And those you can blacklist, see above on the topic of "blacklisting USB networking modules". Keep in mind that the Raspberry Pi 3+ ethernet interface is connected to the internal USB bus. If you blacklist all USB network modules, your ethernet will also be disabled.

As we can't prevent people from plugging in storage and HID devices with blacklisting since modules are compiled in, what other options are there? You can turn off the USB port power on your TurtleBot3. Get the hub-ctrl utility. You will have to compile it from source. If you compile the software on the robot it is best practice to remove the compiler before deploying the robot. By leaving compilers on the robot, an attacker who gained system level access would have the ability to compile malicious software.

Now that you have the binary on your TurtleBot3, for example, to turn off the power on Hub 1 to USB Port 2 and 3 run:

```
jsmith@x86_64:~$ sudo ./hub-ctrl -h 1 -P 2 -p 0
jsmith@x86_64:~$ sudo ./hub-ctrl -h 1 -P 3 -p 0
```

Try plugging in a keyboard or a flash drive; nothing should happen. See the diagram below to determine the location of Port 3 on Raspberry Pi3 +.



Raspberry Pi3 + Port Location

On to the remaining two USB plugs on the left. That's where you should connect the LiDAR and the OpenCR board. The ethernet, OpenCR, and LiDAR will all be on the USB Port 1, and you should physically prevent tampering with the two plugged cables. This is where another limitation of Raspberry Pi3 + comes into play. You can not turn off the power with hub-ctrl to USB Port 1, because for that to work, no devices can actually be plugged into the port when you turn off the power. If a device is plugged in, the USB system will just reinitialise the device, so the port won't actually lose power. But it's not possible to unplug all devices on Port 1. The ethernet device is electrically wired internally to Port 1; thus, it prevents the power from being turned off to Port 1. And because port 1 has two USB plugs, those two plugs will always be powered. Therefore, they will require some physical anti-tampering method.

Disabling Internet Protocol v6

By default, all interfaces come up with an ipv6 address. If you are not using ipv6, you should disable it. Not because there is anything wrong with ipv6 but because you want to reduce the number of pathways through which your robot could be attacked.

```
jsmith@tb3:~$ ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group
default qlen 1000
    link/ether bc:a8:a6:fd:43:be brd ff:ff:ff:ff:ff:ff
    inet 192.168.128.46/24 brd 192.168.128.255 scope global dynamic
noprefixroute eth0
       valid_lft 48058sec preferred_lft 48058sec
    inet6 fe80::ae66:f8fd:c277:efba/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

To disable ipv6 add the following lines into "/etc/sysctl.conf"

```
net.ipv6.conf.all.disable_ipv6=1
net.ipv6.conf.default.disable_ipv6=1
net.ipv6.conf.lo.disable_ipv6=1
```

And then run:

```
jsmith@tb3:~$ sudo sysctl -p
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

Verify that ipv6 is gone:

```
jsmith@tb3:~$  ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group
default qlen 1000
    link/ether bc:a8:a6:fd:43:be brd ff:ff:ff:ff:ff:ff
    inet 192.168.128.46/24 brd 192.168.128.255 scope global dynamic
noprefixroute eth0
       valid_lft 48058sec preferred_lft 48058sec
```

If your robot is based on some other architecture, like x86_64, for example, this step is not needed. But in the case of the TurtleBot3 based on the Raspberry Pi 3, paste the lines below in your console. This will make sure that the "sysctl" is executed after every reboot.

```
cat <<EOF | sudo tee -a /etc/rc.local
#!/bin/sh -e
sysctl -p
EOF
sudo chmod +x /etc/rc.local
```

Disabling Bluetooth

Bluetooth is a convenient way to connect devices wirelessly, but there have been exploits in the past against Bluetooth. For example, the BlueBorne exploit doesn't even need to pair with a Bluetooth device or even need the device to be discoverable. If your TurtleBot3 is not using Bluetooth for any function, you should disable it. Paste the lines below in your terminal, the script snippet will find all Bluetooth kernel modules and will add them to the modules blacklist.

```
for i in $(find /lib/modules/`uname -r` -name bluetooth -type d);\
  do find $i -name *.ko | sed 's/.ko$//g' | awk -F/ '{print
"blacklist",$(NF-0)}';\
done | sudo tee -a  /etc/modprobe.d/blacklist.conf
```

Some modules might already be loaded, so a reboot will clear those out, but if you don't want to reboot, you can modify the above script to remove those modules from memory.

```
for i in $(find /lib/modules/`uname -r` -name bluetooth -type d);\
  do find $i -name *.ko | sed 's/.ko$//g' | awk -F/ '{print $(NF-0)}' | xargs
rmmod ;\
done
```

Keep in mind that some modules might be in use by others in this list and won't be removed on the first try, repeat the process until all lines say

```
rmmod: ERROR: Module [name] is not currently loaded
```

Disabling core dump

Getting a core dump from an application when it experiences a crash is a great way to debug issues in applications. The size of the core dump can vary widely from application to application; it all depends on the memory footprint of the application. The assumption is that an application would seldom experience a problem, but when it does, it will provide memory contents for the developers to look through. While that assumption is reasonable for development, once you move your robot to production, you could face a Denial of Service (DoS) attack. When a process crashes, it's generally assumed that the process is restarted to restore the particular service. For example, the systemd will do that automatically for you. Suppose an attacker finds a way to trigger this application crash through your exported interface, or worse, if they find a way to script it. They could cause the particular service to crash and dump the core hundreds of times per second, which would fill up your storage. The entire robot then will be subject to all sorts of unpredictable consequences as all services start behaving in unexpected ways due to a lack of space. By disabling the core dump, the attacker can only affect the buggy service and not the entire robot system. To disable the core, add the following lines to "/etc/sysctl.conf":

```
kernel.core_uses_pid = 0
kernel.core_uses_pid = 0
```

And run: sudo sysctl -p:

```
sudo sysctl -p
```

Disabling WiFi

If your production robot is stationary and uses an ethernet connection instead of wireless, you should disable the wireless chip. On the TurtleBot3 running RaspberryPi, you do that by adding following lines into "/etc/modprobe.d/blacklist.conf" and reboot.

```
blacklist brcmfmac
blacklist brcmutil
blacklist cfg80211
```

Disabling ethernet

In the event your production robot uses the WiFi connection instead of the wired, then disable the wired connection. Add the following lines into "/etc/modprobe.d/blacklist.conf" and reboot.

```
blacklist lan78xx
```

Conclusion

As you prepare your robot for production, security shouldn't be an afterthought. Putting an upfront effort can, in the long run, save you resources, and a headache that comes with a security breach. As our world becomes more and more connected, a paradigm shift from "if we become a target" to "when we become a target" warrants a proactive approach to security, and remember, security is not a single on/off switch. Security is many smaller actions that on their own, don't necessarily have a significant impact, but it builds strength in numbers. A large number of breaches are opportunistic. If you put up a certain amount of barriers, the attackers will move on to a weaker target.

Learn more about Ubuntu and robotics here:
https://ubuntu.com/robotics

ubuntu
Delivered by Canonical