IT Club Cyber Defense Meeting Notes

17-November-2022	
Location:	GoDaddy Hiawatha Roasters Conference Room
Present:	Anna, Melissa, Michael, Dan
Missing:	Tristan (robotics meet)

Summary of `ssh`

`ssh` is short for **secure shell**, and it has both server and client components.

There are many `ssh` implementations. We used one of the most popular, called **OpenSSH**. The package we installed for the server component is called **OpenSSH Server**, and the server process that it runs is called `sshd`, for **ssh daemon**.

`OpenSSH Server` also happens to install the `ssh` client program, which is simply called `ssh`. You run `ssh` when you want to connect to another computer that is running `sshd`. If that server's administrator has given you permission to connect to it using `ssh`, then you will be able to use a terminal connected to that other computer, which we call a remote computer.

In this scenario, your computer, running `ssh`, is the client, because it initiates the "conversation" between the two computers by sending a connection request to the other computer. The other computer, running `sshd`, is the server, because it is listening for these requests, and responds when it receives one.

How to install **OpenSSH Server**

```
if you have not recently updated apt's
software catalog, do this first:
sudo apt update

sudo apt install openssh-server
```

That's it - one command will install the SSH Server for us. As mentioned before, installing the OpenSSH Server also installs the `ssh` client automatically.

If you are on a computer and you only want to install the client, you should install the package named `openssh-client`. It will install `ssh` but not `sshd`.

Ways to tell if `ssh` is listening for connections on your computer

There are several ways to determine if a service like `sshd` is running on your computer. This section will recap how to do this with `systemctl`, `netstat`, and `lsof`.

Using `systemctl`

The most basic way to check for a running service on Ubuntu is to use `systemctl`. If you know the name of a service, you can get its status from `systemctl`. You can see in the output below that the `sshd` service is active. The log output at the bottom says it is listening on port 22.

```
christopherl@shellsburg:~$ systemctl status sshd
● ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset:
enabled)
     Active: active (running) since Sun 2022-11-20 19:33:29 CST; 5 days ago
       Docs: man:sshd(8)
             man:sshd_config(5)
    Process: 24187 ExecStartPre=/usr/sbin/sshd -t (code=exited, status=0/SUCCESS)
   Main PID: 24188 (sshd)
      Tasks: 1 (limit: 4626)
     Memory: 2.6M
        CPU: 15ms
     CGroup: /system.slice/ssh.service
             └─24188 "sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups"

Nov 20 19:33:29 shellsburg systemd[1]: Starting OpenBSD Secure Shell server...
Nov 20 19:33:29 shellsburg sshd[24188]: Server listening on 0.0.0.0 port 22.
Nov 20 19:33:29 shellsburg sshd[24188]: Server listening on :: port 22.
Nov 20 19:33:29 shellsburg systemd[1]: Started OpenBSD Secure Shell server.
```

Using `netstat`

One of the tools IT professionals rely heavily on for checking port usage is `netstat`. It can tell you the status of every network connection currently open on your computer as well as every listener that is waiting for network requests. We will use `netstat` to see if any program is listening for network requests on TCP port 22, which is SSH's default listener port.

> **Sidebar: TCP vs UDP**
>
> We briefly discussed some differences between TCP and UDP, which are two of the main protocols in the IP protocol suite.

> TCP is connection-oriented, so it emphasizes things like having all of the data arrive in the same order it was sent, and keeping the connection between both computers open, much like a phone call.
>
> UDP is not connection oriented. It doesn't care about the order in which the recipient receives the packets at all. Each packet might get routed separately, with different delays in arrival.
>
> It's a big topic. Google for more info, or come to office hours! If we have them.
>
> ---
>
> **Sidebar: Ports**
>
> To know which services are supposed to receive which network packets, port numbers are used. They are just another number that gives a network destination that is more specific than an IP address. You could think of this like an apartment building: the IP address of a computer is like the address of an apartment building, and a port number is like the apartment number.

The `netstat` program may not be installed on your computer. If not, you should be able to install it on versions 20.04 LTS or 22.04 LTS of Ubuntu by running `sudo apt install net-tools`.

After you have `netstat` installed, you can use it to list all tcp ports that have processes attached to them that are listening for network requests. You can even have `netstat` tell you what process is listening on each port (you have to use `root` permissions to do this, which is why the following command starts with `sudo`).

```
christopherl@shellsburg:~$ sudo netstat --tcp --listening --programs --numeric-
ports

Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
24188/sshd: /usr/sb
tcp        0      0 localhost:631           0.0.0.0:*               LISTEN
27069/cupsd
tcp        0      0 localhost:53            0.0.0.0:*               LISTEN
478/systemd-resolve
tcp6       0      0 ip6-localhost:631       [::]:*                  LISTEN
27069/cupsd
tcp6       0      0 [::]:22                 [::]:*                  LISTEN
24188/sshd: /usr/sb
```

You can see in the output that process `24188/sshd` is listening on port 22, which is the default ssh port. The local `tcp` address of `0.0.0.0`means that `sshd` is listening on every TCPv4 address that the computer is connected to. Similarly, the `tcp6` address of `[::]:` means that `sshd` is listening on every TCPv6 address that the computer is attached to. Long story short, this computer will take `ssh` connections from any available network connection.

Using `lsof`

I want to also briefly mention the program `lsof` which can show you lists of open files. That might not sound overly impressive until you learn that almost everything in Linux has a representation as a file somewhere, so

this goes beyond normal files. In this case, we can ask `lsof` to show us listening `tcp` ports and it will give us the same information `netstat` did. Again, you need to use `sudo` for this.

```
christopherl@shellsburg:~$ sudo lsof -iTCP -sTCP:LISTEN

COMMAND     PID            USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
systemd-r   478 systemd-resolve   14u  IPv4  17151      0t0  TCP localhost:domain
(LISTEN)
sshd      24188            root    3u  IPv4 171444      0t0  TCP *:ssh (LISTEN)
sshd      24188            root    4u  IPv6 171446      0t0  TCP *:ssh (LISTEN)
cupsd     27069            root    6u  IPv6 232881      0t0  TCP ip6-localhost:ipp
(LISTEN)
cupsd     27069            root    7u  IPv4 232882      0t0  TCP localhost:ipp
(LISTEN)
```

> SIdebar: Port Numbers and Names
>
> By default, programs like `netstat` and `lsof` use names like `ssh` instead of numbers like `22` to identify ports. For example, `sshd` looks something like this in default `netstat` output:
>
> You might have noticed that, by default, `netstat` says that my `sshd` server is bound to all local IP addresses for the `ssh` service. That looks like this in the `netstat` output:
>
> ```
>    tcp        0      0 0.0.0.0:ssh             0.0.0.0:*
> LISTEN      24188/sshd: /usr/sbin/sshd
> ```
>
> However, when I use the --numeric parameter with `netstat`, `ssh` is replaced with the number `22` so that the output looks like this:
>
> ```
>    tcp        0      0 0.0.0.0:22              0.0.0.0:*
> LISTEN      24188/sshd: /usr/sbin/sshd
> ```
>
> Read on for more information about these service names and numbers.

Ways to tell what other services are listening for connections on your computer

The default port assignments of "well-known" ports are stored in a file named `/etc/services`. For example, if you use the search utility `grep` to look in this file for the string `ssh`, you get this result:

```
grep ssh /etc/services
ssh      22/tcp          # SSH Remote Login Protocol
```

This tells you that the default usage of TCP port 22 is reserved for SSH. This file is informational only. There is no enforcement of the information in this file. There is nothing stopping another service from using TCP port 22 if it wants to, for example. I believe that this file is where `netstat` and other programs get the labels they use when describing ports.

You can use `grep` to search `/etc/services` for specific service names or port numbers. This is a good way to try to identify processes that are listening on other ports. Recall the output we received from `netstat` earlier

```
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
24188/sshd: /usr/sb
tcp        0      0 localhost:631           0.0.0.0:*               LISTEN
27069/cupsd
tcp        0      0 localhost:53            0.0.0.0:*               LISTEN
478/systemd-resolve
tcp6       0      0 ip6-localhost:631       [::]:*                  LISTEN
27069/cupsd
tcp6       0      0 [::]:22                 [::]:*                  LISTEN
24188/sshd: /usr/sb
```

We already know that port 22 is `ssh` from earlier discussions. But what are ports 53 and 631? Are those processes I want my computer to be running?

Let's check whether those ports are listed in `/etc/services` and what their names are if they are listed there. I'm going to use a little bit of advanced search syntax, telling `grep` to search using the pattern `\b(631|53)\b`. In English, the means that it's looking for the strings "631" or "53", but they have to be whole words, meaning that strings like "10631" or "53abc" won't match.

```
grep -E '\b(631|53)\b' /etc/services
domain      53/tcp          Domain Name Server
domain      53/udp
ipp         631/tcp         Internet Printing Protocol
```

Port 53 is reserved for DNS, the Domain Name System, on both UDP and TCP. Both `netstat` and `lsof` have reported that process ID (PID) 478 owns this port. I can use `ps` (process status) to look at PID 478. If I do, I see this.

`ps` parameters used

- The -f parameter makes `ps` use "full" (wide) output, displaying more information.
- The -p parameter makes `ps` only display information for the process ID it is given.

```
christopherl@shellsburg:/proc/24188/net$ ps -f -p 478
```

```
UID          PID    PPID  C STIME TTY          TIME CMD
systemd+     478       1  0 08:45 ?        00:00:00 /lib/systemd/systemd-resolved
```

These results from `ps` look like this could be a service (notice the **d** at the end of the process name?). Let's see if **systemctl** knows about this process.

When I typed the command below, I typed something like `systemctl status systemd-r` and then hit tab. When this didn't autocomplete, I hit tab twice, to see what the options were. That tab key is your friend.

```
christopherl@shellsburg:/proc/24188/net$ systemctl status systemd-resolved.service
● systemd-resolved.service - Network Name Resolution
     Loaded: loaded (/lib/systemd/system/systemd-resolved.service; enabled; vendor
preset: enabled)
     Active: active (running) since Thu 2022-11-17 19:33:06 CST; 3 days ago
       Docs: man:systemd-resolved.service(8)
             man:org.freedesktop.resolve1(5)
             https://www.freedesktop.org/wiki/Software/systemd/writing-network-
configuration-managers
             https://www.freedesktop.org/wiki/Software/systemd/writing-resolver-
clients

   Main PID: 478 (systemd-resolve)
     Status: "Processing requests..."
      Tasks: 1 (limit: 4626)
     Memory: 8.5M
        CPU: 465ms
     CGroup: /system.slice/systemd-resolved.service
             └─478 /lib/systemd/systemd-resolved
```

So yep, that looks like a DNS-related service. They even put URLs in there for me to find out more about their service.

Here's a question that might be very hard for you at this point in your learning. I will try to remember it for our next meeting but I'm not sure I will. It's nice to be able to pull up all of this information, but *how do we know we're not just looking at very convincing malware* that just looks a lot like a service that it's impersonating?

How to make Very Wise Cows(TM)

Now we began to get a little hands-on experience installing programs. Why start with practical ones when we can make Very Wise Cows(TM) before Elon Musk invents them?

The `fortune` command spews out bits of time-tested wisdom, much like fortune cookies. In other words, don't trust anything it tells you. Run the following

```
sudo apt update
sudo apt install fortune
fortune
```

```
fortune
fortune
fortune
fortune
heh
```

The `cowsay` command lets you make cows say things. This clearly protects your anonymity, since nobody will suspect that the text you entered on your workstation could possibly have come from you if a cow is clearly the one saying it.

```
sudo apt install cowsay
cowsay $USER did not tell me to say anything
echo I am a sentient cow | cowsay
fortune | cowsay
```

> **Sidebar: Pipelineing**
>
> The last two lines above use something called pipelines. A pipeline has two or more commands separated by vertical pipes. From left to right, each command is run, and the output of each command is sent to the following command as input. In the example about the sentient cow, the `echo` command just outputs whatever you tell it to. I know - not the most exciting command in the world. So in this case, it outputs "I am a sentient cow" but instead of displaying that on your terminal, as it normally would, it "pipes" that output to `cowsay`. `cowsay` has been written so that if it receives input in this manner, then the cow will use that as its text.
>
> The last line, the one that runs `fortune`, does the same thing, but instead of you hard-coding a message that `echo` will pass to `cowsay`, you use the `fortune` command to generate the text for the cow to say, thus producing a Very Wise Cow(TM).
>
> For what it's worth, this is also the line that really convinces people that the cow is the one doing all the thinking. There's no way I'd come up with such elegant sayings, and all my friends and family know it. I've had to dissuade them more than once from forming The Church of Unworthy Scripters in Service of the Wise Order of Cows(TM). I usually do this by pointing out that this could be abbreviated as The Cussword Cows. They really don't like that, as it is clearly blasphemous, so the conversation typically ends there.

You can also specify different pictures to use instead of the Very Wise Cow(TM). This would probably cause The Cussword Cows to form splinter groups and have factional disputes and argue over how many `cowsay` avatars can dance in a Thunderbolt port or something.

To list the "animals" that are available, use the `-l` parameter, and to use a different animal, use `-f` (for cowFile). There are a few other parameters. As always, Google or `man cowsay` is how you find this info.

If for some reason you wanted to use `cowsay` with all of its animals except `stimpy`, because you love `ren` and think `stimpy` is always stealing the spotlight, this is one way you could do that.

```

```

```
for animal in $(cowsay -l | tail +2); do
    if [[ "$animal" == "stimpy" ]]; then
        echo "$animal" is a showboat
    else
        fortune | cowsay -f "$animal"
    fi
done
```

If you want to gain insight into how and why this works, the way to do that is to run the commands individually, building up the complete `for` loop from its various parts.

Let's start with the first line. The first part that has to be run is the part inside the parentheses, just like in math.

The pattern `command2 $(command1)` is very similar to the pattern `command1 | command2`. In both cases, `command1` is run, and its output is passed to `command2`. What is different is that with the `$(...)` syntax, the output goes exactly where you tell it to go, instead of being piped to the `command2` and assuming that `command2` will know what to do with it. In this case, we are explicitly putting the output of `cowsay -l | tail +2` immediately after the `in` keyword on line 2.

That's a lot of words. Let's look at how this works. Please follow along in your terminal.

Too see all the animals that `cowsay` can use:

```
this produces a list of all of the animals `cowsay` knows about:
cowsay -l
Cow files in /usr/share/cowsay/cows:
apt bud-frogs bunny calvin cheese cock cower daemon default dragon
dragon-and-cow duck elephant elephant-in-snake eyes flaming-sheep fox
ghostbusters gnu hellokitty kangaroo kiss koala kosh luke-koala
mech-and-cow milk moofasa moose pony pony-smaller ren sheep skeleton
snowman stegosaurus stimpy suse three-eyes turkey turtle tux unipony
unipony-smaller vader vader-koala www
```

We have no use for the first line of output - the one that says, "Cow files in /usr/share/cowsay/cows:". We can use the `tail` command to skip the first line. Run `man tail` to see why this works.

```
cowsay -l | tail --lines=+2
apt bud-frogs bunny calvin cheese cock cower daemon default dragon
dragon-and-cow duck elephant elephant-in-snake eyes flaming-sheep fox
ghostbusters gnu hellokitty kangaroo kiss koala kosh luke-koala
mech-and-cow milk moofasa moose pony pony-smaller ren sheep skeleton
snowman stegosaurus stimpy suse three-eyes turkey turtle tux unipony
unipony-smaller vader vader-koala www
```

So far, so good. We now have a list of all of the `cowsay` animals. Now we just need to find a way to look at the names, one at a time, so that we can tell whether the current one is `stimpy` or not.

The bash shell has a built-in `for` loop that lets us do this easily. For example, the following loop creates a variable named $person that is given the students' three names, and then simply displays them by using the `echo` command.

```
for person in Anna Michael Tristan; do
    echo "$person"
done

Anna
Michael
Tristan

You can do anything you want with a for-loop variable. For example, you can use
this variable to make the `cowsay` messages more personalized. It's very
convincing.

for person in Anna Michael Tristan; do
    echo "$person, " $(fortune) | cowsay
done


 ________________________________________
/ Anna, Q: What do you get when you cross \
| the Godfather with an attorney? A: An   |
\ offer you can't understand.             /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


 ________________________________________
/ Michael, Your life would be very empty \
\ if you had nothing to regret.          /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||


 ________________________________________
/ Tristan, Chicken Little only has to be \
\ right once.                            /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

In our case, for each time our $animal variable changes, we want to test if its value is "stimpy" or not. The `bash` shell gives us an easy way to do this also, by providing an `if` statement. there are many things an `if` statement can check, but the most common checks are to see whether two values are equal That is exactly what we want to do.

```
for animal in ren stimpy; do
    if [[ "$animal" == "stimpy" ]]; then
        echo "$animal is a showboat."
    else
        echo "We like "$animal" a lot, even if we are not sure why."
    fi
done

We like ren a lot, even if we are not sure why.
stimpy is a showboat.
```

If all of these little pieces make sense to you, then you have all the concepts down to understand our original for loop also. If some things aren't making sense just yet, that's OK. Keep studying examples as you have time and keep asking questions. The amount you get better each year depends on the amount of time you put in, being both eager and humble enough to ask the team for help when you need it, and having fun 😃.

Here's that original `for` loop:

```
for animal in $(cowsay -l | tail +2); do
    if [[ "$animal" == "stimpy" ]]; then
        echo "$animal" is a showboat.
    else
        fortune | cowsay -f "$animal"
    fi;
done
```

And now that you've read all of this, here is a concise explanation of how it works.

section or keyword	**function**
for	Introduces the for loop.
animal	The name of your for-loop variable. note that when you create or set a variable in bash, you do not use a dollar sign at the beginning of its name. when you read its value, you do.
in	Introduces the command or list from which you will get values for your for-loop variable.
$(...)	Whatever is in this location creates the values that the for-loop variable will iterate over. It does **not** have to be in the form `$(...)` - it can be any valid expression that returns a list of values.

section or keyword	function
do	This marks the end of the loop header and means that the loop body follows.
loop body	Everything between the keywords **do** and **done** are the loop body. Each time there is a new value for the for-loop variable, the loop body executes for that new value. In this particular case, the loop body is an `if` statement that determines whether the $animal variable is equal to "stimpy" and responds accordingly.
done	This means the loop is complete. At this point, the loop tries to get another value for the for-loop variable. If it can get a new value, it performs whatever logic is inside the loop again. if there are no more values available for the for-loop variable, the loop terminates.

> **Sidebar: Code of Conduct in Open-Source Software** At our meeting, I was concerned about one of the "animals" on the list, but as Dan reassured me, it turned out to be a chicken. Lest you think I'm just a paranoid old man, let me tell you why I think this.
>
> The Linux development community has had its share of dysfunction, especially in the past. Its founder, Linux Torvalds, was one of the well-known offenders. He treated people poorly for various reasons, such as letting them know if he thought their ideas were stupid. During this time, `cowsay` and a number of other programs did, in fact, have offensive material in them.
>
> A few years ago, and strong push was made to include a Code of Conduct in the Linux project. Unsurprisingly, some folks were upset about this, thinking silly things like somehow being nice to each other would mean that the project would have to accept bad code. I guess some people don't actually know how to tell somebody else that their code isn't ready to be accepted or that their feature request won't be implemented without insulting them.
>
> Anyway, the code of conduct was adopted; Linus has to be nicer now; and the offensive content from programs like `cowsay` and `fortune` has apparently been removed.

Homework

If you have the time and you'd like to try something new that will reinforce a lot of things in this document, here are some homework ideas. Do as many of these as you have time for. Please ask questions in either #homework or @safe_place if you need help so that the team can get used to helping each other. This will be important for the competition.

Background

There is a file called `/etc/passwd` that contains one line for each user on an Ubuntu computer. The file format might look strange at first, but it is really a bunch of fields separate by colons. Here's the entry for my user account on one of my VMs:

```
christopherl:x:1000:1000:christopherl,,,:/home/christopherl:/bin/bash
```

The first field is my username, `christopherl`. The second field, `x`, is there for historical purposes, but basically means that the hash for my password is not stored here. There are some other fields that have to do with my user ID, group ID, and group name, but then I want to call attention to the last two fields. `/home/christopherl` is my home directory, and `/bin/bash` is my default shell, meaning that if I log onto this computer, unless another shell is specified specifically, I will be given a bash shell by default.

Here are some things I'd like you to try to do with `/etc/passwd`. For each question, I will give you the name(s) of shell commands that you could use to do the task. To learn how to use these commands, use Google or `man`.

Tasks

1. How many users are listed in `/etc/passwd`? Don't count them yourself; use bash to count them for you.
    1. You may assume: every line in `/etc/passwd` corresponds to 1 unique user
    2. Things that could help: the `wc` command
2. Can you run a command that will cause `bash` to print only the line for the user `whoopsie`?
    1. Things that could help: `grep`
3. How many users have `/bin/false` listed as their default shell?
    1. Things that could help: `grep` and `wc`, used together (think pipelines)
    2. FYI: the default shell is the 7th field for each line in `/etc/passwd`. I'm not saying you need to use this information, but if you want a 100% certain answer, it might be ueeful.
        1. Given this comment, other things that might be useful include `sed` and `awk`, but those are overkill, and I'd just use `cut`. I don't expect anybody to figure out anything I'm saying about field number 7. 😃
4. What is your home directory? Can you get `bash` to display it by itself, without the rest of the line from `/etc/passwd`? For example, if it was me, I'd want the result to just say `/home/christopherl` but not to have any of the other parts of the line.
    1. Things that could help: `grep` and `cut`. You will want to pay attention to the `-t` parameter of `cut`.