Problem Set 2

Joe Puccio

October 22, 2015

Prelude

Began by cloning the existing prebuilt VM that was provided running Ubuntu 12.04. This resulted in two machines with identical IP addresses on all interfaces (confirmed by running ifconfig | grep inet), which means they were not able to communicate with each other. I solved this by generating a host-only interface in Virtual Box for both VMs, being sure to assign different MAC addresses to each of the VMs. I verified success by telneting into each of the machines from the other one, and creating a file on the desktop to verify I was connected to the correct machine. I then made sure that I was able to connect to the internet on both machines, and I did so by both pinging www.google.com and visiting it in Firefox on both machines. Then, I touched files on each of the desktops of the machines indicating which I would then on refer to as "A" and which "B".

2.1

The first thing I did was, by default, allow incoming traffic (rather than drop) by running "ufw default allow incoming" with root permission (as sudo). I verified that I had altered the "DEFAULT_INPUT_POLICY" parameter by running "cat ufw | grep DEFAULT" (piping the contents of ufw to grep), and checking that the input policy line was set to "ACCEPT", as required. With this set to accept on both VMs, they will by default accept all incoming traffic rather, as the prior setting was to default drop all incoming traffic. In order to avoid having to repeatedly escalate to root user, I ran "su" and typed in the root password. After flipping through the manual page for ufw (running "man ufw", naturally), I started by running "ufw status" to check the status of the firewall. This revealed that the firewall was not active, so I activated it by running "ufw enable". I then verified that I was still able to telnet between machines.

The manual page makes it clear that using ufw makes setting up rules really simple: it will even look in /etc/services to see if your rule is specifying an application and determine which protocol (TCP or UDP) to block on which port. But no need to have it do the extra work, so we'll just specify 23/tcp directly to block telnet between machines (because telnet runs over tcp, by default on port 23). To prevent machine A from telnetting into machine B, we may run "ufw deny 23/tcp" on machine B (we could have equivalently done "ufw deny in 23/tcp", but when direction isn't specified, inbound traffic is assumed). To verify the rule is in place, we may run "ufw status numbered", which reveals all of the rules we've set in place (it appears as though two rules were created by this command, one for IPv4 and one for IPv6). Running "ufw reload" doesn't appear to be necessary for the rules to come into effect. After adding this rule, attempting to telnet from machine A into machine B resulted in a hang, and then finally a timeout. Adding this rule to machine A had the same result when attempting to telnet from machine B. These rules can be deleted to clean up by running "ufw delete [*number*]", where "number" is the number of the rule.

Of course, it should be noted that blocking TCP traffic on the default port of 23 will not guarantee all telnet traffic to be blocked. An individual in control of both hosts would be able to configure which port the telnet service listens to, potentially pointing it to a port other than 23, and then connecting via telnet to that port by specifying the new port as an extra parameter.

I prevented machine A from visiting my website UNC Class Finder, www.uncclassfinder.com, by first running “dig www.uncclassfinder.com” to recall the IP of my server (the domain points to only one server), and then running “ufw deny out to 162.243.230.190 port 80”, which prevents machine A from performing the TCP handshake with my server after it has resolved the IP from the domain. This was confirmed as the page would hang after trying to visit www.uncclassfinder.com in Firefox.

2.3

Protip regarding tunneling: paid airplane WiFi usually lets DNS requests go through for free, which means DNS tunneling is possible (HTTP over DNS). You just have to have a server setup on the internet beforehand running the right service before takeoff (I recommend Iodine).

(Keep in mind the note about blocking telnet traffic in my previous answer.) I was able to block all outgoing traffic on machine A to external telnet by running “ufw deny out 23/tcp”, which blocks outgoing traffic to tcp port 23. After adding this rule, the telnet connection to server B timed out.

Rather than use Facebook.com as the website to block, I decided to use my website again which I know is served by only one server (only one IP to block). Again, to block my website I ran “ufw deny out to 162.243.230.190 port 80” and confirmed that it was blocked by attempting to visit www.uncclassfinder.com via Firefox (the page would not load, of course the domain can still be resolved though). Note that my site does not use SSL, so clients connect to it via HTTP. This is the reason I specify port 80 to be blocked, otherwise it may be necessary to block TCP on port 443 because HTTP over SSL (HTTPS) uses TCP on port 443. Also, again, the port choice of a web server is arbitrary/a convention and I could have specified my webserver to also host the website on port 1080 for example, if I wanted to, in which case TCP on port 1080 would need to be blocked as well. Browsers assume you’re attempting to connect to the server on port 80 unless you explicitly specify the port (e.g. www.uncclassfinder.com:1080)

To create the SSH tunnel from machine A to machine B, I executed “ssh -L 8000:192.168.56.102:23 seed@192.168.56.102” on machine A, where “192.168.56.102” is the IP of machine B. This opens a socket at port 8000 on the local machine, A, (localhost), and whenever a connection is made to port 8000 on localhost, the connection is forwarded through machine B inside the SSH tunnel (therefore bypassing the firewall, because the traffic looks like normal SSH traffic, TCP on port 22, to the firewall rather than telnet traffic, TCP on port 23). We actually make this connection by opening up a new terminal tab on machine A and typing “telnet localhost 8000”, and we see that we are successful as we have made an outgoing telnet connection on a machine that is blocking outgoing TCP 23 traffic. By packet sniffing with Wireshark, I observed on the wire that the only packets that ever go out to machine B from machine A are TCP port 22 (SSH traffic), even (and most importantly) after running “telnet localhost 8000” and running commands in the telnet session. Also, importantly, all of the datagrams were encrypted because they were being tunneled through SSH, which is a plus because telnet is all transmitted in plaintext. Because this method is using static port forwarding, the tunneled traffic will be sent to machine B on port 23, regardless of whether the traffic should be sent to that port. For instance, I believe running “ftp localhost 8000” would fail, because the machine B is not going to be listening for the ftp protocol on port 23. However, if we run “ssh -L 8000:192.168.56.102:**21** seed@192.168.56.102” on machine A, and then ran “ftp localhost 8000”, then we would be able to potentially connect to an ftp server on machine B.

To create the SSH tunnel to machine B through which we’ll reach our blocked HTTP server (www.uncclassfinder.com), we can run “ssh -D 9000 -C seed@192.168.56.102”. This uses dynamic port forwarding, which is preferable especially because we may not know necessarily if the server we’ll be attempting to connect to will want us to connect on port 80 (HTTP) or 443 (HTTS). In our instance, it doesn’t matter because I know my web server only serves on port 80, but doing dynamic forwarding is generally preferable for this reason if you're trying to tunnel general web traffic. This command opens a socket on port 9000 on localhost, which allows us to shove traffic through it, and the traffic will be forwarded to machine B through the SSH tunnel (thus bypassing the firewall), and machine B will be making the web request for us, take the response, and shove it back through the SSH tunnel (bypassing the firewall) to us (machine A). Again, the firewall is bypassed because it's only blocking outgoing connections from machine A to the server for www.uncclassfinder.com, but using tunneling we're making it look like we're just sending and receiving some regular-ol' SSH traffic with our friend machine B. We then point Firefox to use localhost port 9000 as a proxy (specifically SOCKS, which I believe will transmit both UDP and TCP data, so perhaps DNS requests are going through the SSH tunnel), which means it'll send and receive all requests through it. After configuring the proxy in Firefox, we are able to connect to the previously blocked www.uncclassfinder.com. The page fully loads. After breaking the tunnel and clearing the cache, Firefox complains that the proxy is refusing connections, naturally, because no service is running on port 9000 on localhost anymore. After re-establishing the SSH tunnel again, www.uncclassfinder.com loads. To reiterate, the SSH tunnel helps bypass egress filtering because it disguises traffic that would be blocked (such as telnet traffic, or traffic to a specific IP in our examples) as whitelisted traffic (such as SSH, in our example). It relies on having control of a server outside of the firewall, though. We can see again by looking at the packets on the wire that all packets sent from machine A, the machine behind the firewall, are TCP port 22 destined for machine B, which is whitelisted traffic, so it won't be blocked.

If ufw were to block TCP port 22, you would not be able to evade egress filtering using an SSH tunnel because the SSH tunnel is operating on TCP port 22. In that case, it'd be best to evaluate which ports are open for you to use and tunnel through them. For instance, if port 53 (DNS) is open, you can tunnel your web traffic through port 53.

2.4

The first thing I did was clean up from the previous exercise, mainly just by running "yes | ufw delete 1" a few times until all the rules were gone on machine A, and then resetting the network settings in Firefox. Next, I navigated to /etc/squid3, as Squid was already installed on machine B, and took a look at the config file in there. Then, I pointed machine A's Firefox to use machine B as its proxy, pointing it specifically to port 3128 on machine B, where the squid process listens for its prey.

Visiting sites on machine A reveals that all http traffic is being blocked by squid, however all https traffic is not being blocked by squid. This is because by design, the http request is encapsulated in a layer of encryption (SSL) and so squid is unable to regulate this traffic intelligently. This is a disadvantage of using application layer filtering. By examining the squid.conf file, we can see that all http traffic was being blocked by default as there were no user set rules and thus the only rule that applied to them was the last rule, the default rule, "http_accces deny all", and thus these requests were blocked. We may allow all websites by inserting "http_accces allow all" anywhere before that deny statement. This allowed access to all http sites. We may allow access to *.google.com by creating an access control list (ACL) named "googleTraffic" with the type "dstdomain" and the definition ".google.com", and then allowing http access to that ACL, as so:

```
acl googleTraffic dstdomain .google.com
http_access allow googleTraffic
```

And these lines just have to go somewhere above the catch-all "http_accces deny all" line. Examining Wireshark confirms our postulates above: the squid proxy server is actually examining the content of the requests being made by the client, and running them through its ACLs to see if the request should be carried out, or the request should be blocked. The destination for all of the traffic coming from machine A (packets) is the proxy server, naturally, because Firefox is configured to send all of its traffic through the proxy server who will make (or not make) the requests on behalf of machine A and return the result.

We can use a web proxy to evade egress packet filtering because if we configure our browser to route our traffic through a machine which is outside the packet filtering firewall and if in fact traffic to that machine is not blocked by the firewall, then the firewall be perfectly happy allowing traffic to our proxy who will make requests to potentially blocked servers/ports on our behalf, and return us with the response. This is done by first setting up ufw on machine A to say, block access to the IP of www.uncclassfinder.com on port 80 (or all ports; won't make a difference). Then we setup a web proxy (such as squid) on machine B which is configured to at least allow access to www.uncclassfinder.com, if not all sites. Then we configure Firefox on machine A to use machine B's squid instance as a proxy (as we did before, by specifying machine B's IP and the default squid port), and then requesting www.uncclassfinder.com from Firefox on machine A returns the page, because to the egress packet filter, it appears as though no traffic was sent to (or even from, so ingress filtering is no good either) the IP of www.uncclassfinder.com, only to machine B's IP. The entire idea here in all of these circumvention methods is to shuffle the revealing information about who in fact you are talking to up and down the OSI stack, to wherever the evil censors aren't listening.

If ufw blocks the TCP port 3128, you can still use a web proxy to evade the firewall. In fact, you can still use squid to evade the firewall. All you need to do is configure squid to operate on another port, one that's not blocked by the firewall. Or, you can use another web proxy that just by default does not run on port 3128.

The perl program "myprog.pl" is run against every URL requested by clients using that proxy server. The program attempts to match the URL requested by the client against exactly "www.cis.syr.edu", and if there's a match, it will rewrite the request URL to yahoo.com, which means the proxy server will request yahoo.com instead and return the response from Yahoo instead of requesting www.cis.syr.edu and returning its response. The program appears to flush the output buffer on every print in order to handle requests as soon as possible. Only the "www.cis.syr.edu" URL got rewritten to Yahoo, while all other URLs were not rewritten.

To redirect all Facebook pages to a big red stop sign, we can modify the URL check line to $url =~/.*\.facebook\.com/, and then to change the url we're redirecting to, we change the print line inside the if statement to: print "https://www.reloadit.com/Content/images/stop-sign.png". I had trouble getting this working with the proxy, that is when visiting a facebook page in the browser I wasn't able to force a rewrite, even though everything checked out in the terminal when piping strings to the file, that is, "echo "app.facebook.com" | myprog.pl" would print the stop sign URL.

To redirect all gifs and jpg images to an image of our choice, can change the URL check line to $url =~/.*\.jpg | .*\.gif/. This will match any request that has the file extension .jpg or .gif. I was able to get this working with the proxy. We also change the print line inside that if statement to print "https://media0.giphy.com/media/lk0TFUdop2JTW/200_s.gif", which is Admiral Ackbar's "it's a trap!" gif.