CS6740 Project 1

Andrew Barba abarba@ccs.neu.edu

Report

My report is broken up into subfolders for each task, ex: `./task_1` contains instructions, code, screenshots, and any findings/results associated with Task 1. Each subfolder will contains a `README.md` with final results and findings for that task. This document will contain any global findings and useful setup information.

Submission

My final report is in a single PDF `./REPORT.pdf` which was generated from `./REPORT.md`. `REPORT.md` was built using a simple bash script to compile all the reports in each task subfolder.

Git

This folder lives in a private Github repository which was then cloned onto each VM. This allowed me to share common setup scripts between the machines. It also allowed me to develop in a single, familiar environment (Atom.io on Mac OS X), and then simply pull down changes in each VM.

Task 1

Description

This attack attempts to inject an entry in the ARP cache of a victim. This is useful for an attacker because once an entry is in the cache the MAC address for a specific IP does not need to be looked up again. An attacker may choose to alter the associated MAC address for a database server, preventing the victim from connecting that database and potentially bringing it offline.

Evidence

All evidence for this attack can be seen in the `./task_1/screens` folder.

Code

This attack was executed using a single command which can be seen in `./task_1/code/poison.sh`.

Notes

When I first tried executing the `netwox` command I was getting an error saying the victim IP was `unreached`. Thanks to Aboobacker Rizwan, I realized it was necessary to specify the device in the command because `netwox` was using the NAT interface by default.

Task 2

Description

ICMP redirects are used by routers to tell systems to update their routing tables for a specific IP address. Because there is no authentication an attacker can use this to send a valid ICMP redirect packet to a host and modify the record of an IP address to and address of their choosing. This can be used as a denial of service attack to prevent a host from reaching a specific IP.

Evidence

In `./task_2/screens` you can see the attacker using netwox to send an IMCP redirect packet in an attempt to take redirect from `192.168.56.101` to an invalid IP `192.168.56.133`. You can also see the packets being sent in the Wireshark screenshots.

Code

This attack was executed using a single command which can be seen in `./task_2/code/icmp_redirect.sh`.

Notes

Ultimately I could not redirect traffic away `192.168.56.101` because all of the IP's are under the same LAN and the hosts were not going to the router to look up routing information. If they were, we would see in the `route` command that the address would be poisoned with the invalid IP address.

Task 3

Description

SYN flooding is an attempt to fill up a victims half-open connection queue by sending the first packet in a typical 3 packet handshake, and then continue sending SYN packets without ever completing the handshake. As the queue fills up on a certain port, that port will become unreachable.

Evidence

I ultimate could net get this attack to work using either Netwox or Scapy. When attempting to use Netwox, the victim would simply never reply with a SYN-ACK. After reading thread on Piazza I switch to Scapy and began sending packets in a loop, this successfully began receiving SYN-ACK's but I could never DOS the victim. I started a simple Node.js server listening on port 8081 and even though Scapy was sending packets to the same port and the victim was acknowledging the packets,

the server was always reachable. I checked multiple times to make sure syncookies were disabled on the victim but that did not seem to help.

Code

Netwox:

```
netwox 76 -i 192.168.56.102 -p 80
```

Scapy:

```
send(IP(dst="192.168.56.101")/TCP(flags="S", dport=8081), count=1000000)
```

Notes

Screenshots of all the failed attempts can be seen in `./task_3/screens`.

Task 4

Description

A TCP packet has a reset flag, RST, which can be set to indicate that a TCP connection should be terminated immediately. This can be useful in a lot of situations, such as a server being under load and indicating that a client should stop connecting. But it cal also be used by an attacker to prevent legitimate connections.

Evidence

Evidence for this attack can be seen in three screenshots in `./task_4/screens`. There is a screenshot of the attacker issuing the `netwox` command, the victim unable to connect to the other host on the network, and a screen showing the RST packets in Wireshark.

Code

A single `netwox` command was issued from the attacker to the victim, flooding the victim with TCP RST packets. This command can be seen in `./task_4/code`.

Notes

I originally wanted to complete this task using scapy but for whatever reason, could not get the attack to be successful. I tried many different variations of the arguments: `sport`, `dport`, `ack`, and `seq`. The command that I felt should have worked, and did not, was:

```
send(IP(dst="192.168.56.101",src="192.168.56.102",ttl=128)/TCP(flags="RA",sport=23,
dport=52250,ack=1730944919,seq=0), loop=1)
```

This command should have sent an RST ACK packet to the victim's telnet port with the appropriate ack number. I looke dup the appropriate values using Wireshark and replicated exactly what netwox was sending in the successful attack. I a not certain why this command failed.

Task 5

Description

Task 5 is similar to Task 4 in that the attacker will attempt to upset a network connection by flooding the victim with RST packets. The big difference in task 5 is attempting to disrupt a video stream. The assignment specifically states to try and disrupt a stream of a common video network but because of our setup, we must stream the video in the LAN. For this we installed VLC media player to read a stream and play video, and I chose to use Node.js to install a simple http server to stream the video to the client. The attacker then floods the victim with RST packets in hopes of disrupting the stream.

Evidence

The attack was successful as seen in the `./task_5/screens` directory. The victim initially started playing the video stream, and then after a couple seconds I initiated `netwox 78` from the attacker VM. This immediately started disrupting the stream to the point that VLC stopped retrying and displayed an error on the screen.

Code

Three main code files were used in this task: `httpserver.sh` used to stream the video files, `vlc.sh` used to download and start the VLC media player with a given stream, and `rst.sh` which was the actual attack. All code files can be seen in the `./task_5/code` directory.

Notes

In this task I chose not to try scapy because of the difficulties I experienced in Task 4.

Task 6

Description

This attack again takes advantage of the trust policy in TCP, where a client just assumes that if it gets a packet with a certain bit of info that it can act on that info. In this case we are triggering a "hard error" code to be sent to the victim in an attempt to break the current connection.

Evidence

I ultimately could not get this attack to be successful. The best I was able to do was stall the initial `telnet` connection so it would never prompt the victim to login and the connection would essentially be unusable. However, if the victim was able to login and complete the connection, I could not interrupt it as the attacker. The victim could list files, run commands etc. and would only exit if they chose to do so.

Code

The code for this attack was a single `netwox` command:

```
netwox 82 -i 192.168.56.101 -c 2 -d eth17
```

Notes

The `netwox` command accepts an error code parameter and I tried many variations of the parameter but all failed to interrupt the connection.

Task 7

Description

In Task 7 our goal was to observe TCP (Telnet) traffic between a victim and an observer; calling them M1 and M2 respectively. An attacker, M3, looks to hijack their session traffic and run arbitrary commands on M1's, the victim, machine.

To do so, the attacker first poisons the ARP cache of both M1 and M2, telling M1 that M2 lives at the attackers MAC address, and telling M2 that M1 lives at the attacker's MAC address. By poisoning both caches, M1 and M2 will be able to communicate back and forth, but now M3 has become a man-in-the-middle and can observe all traffic between the two machines. As traffic is sent from M1 to M2, we attempt to forge legitimate packets with our payload in an attempt to get M2 to run the command and report back to M1 it's results.

Evidence

Evidence for a successful ARP poison can be seen in `./task_7/screens`. In `attacker_sniff_password.png` you can see that the attacker successfully sniffed what M1 typed and sent to M2. Other screens show attempts by M3 to forward the command `echo "Hello, World"` however no attempt was successful in getting one of the machines to actually run the command. My guess is I was no properly replicating the sequence and acknowledgment numbers when forwarding the packets. I tried simple things like increasing sequence numbers by 1 but ultimately could not figure out why the attacks were not successfully being run.

Code

There are two parts to the code for this attack. `./task_7/code` has two files: `hijack.py` which continuously forwards ARP requests to the victim and observer in to remain a man in the middle. `poison.py` attempts to do the actual packet sniffing and forging.