Tyler Jackson
2/26/2015

Advanced Network and System Security Assignment 2

For this assignment we were instructed to create a “covert” channel. A covert channel can consist of any means of sending data over a medium in a way that it was not originally intended. It is also implied that the data will be not easily seen or obtained from a normal user. In other words A user would send their own data over the medium while you send your own alongside it and the user has no idea. This can be done in several different ways. Some of the more traditional ways involve hiding the data in the IP identification header. This isn’t typically used, and can therefore be replaced with little notice. The way one could send a message through this channel would be to send a couple bytes at a time with each packet. This way anyone who was observing a packet wouldn’t see much of a semblance of anything even if they were looking directly at the IP ID header. A user would have to look at a collection of packets together to have any reasonably chance of catching an attacker. This attack could be implemented with any stack that uses IP.

Another traditional covert channel is created through manipulating the timestamp of a TCP handshake. Because TCP uses a three way handshake, each packet has a timestamp and sequence number so as to keep the handshake in order in the case of dropped or lost packets. If a certain amount of time has passed and no ACK has been received then a new SYN packet could be sent. The lower order bits of a timestamp are usually somewhat randomized and can therefore be manipulated easily without detection to the naked eye. Therefore, similar to the IP ID field we could slowly leak information through the channel.

Another implementation would be with a stateless protocol like UDP in which the source fields are less important as no acknowledgement is required. This can be used for example if a government was monitoring all incoming traffic on certain ports one could send traffic out without having to worry about getting a response and getting caught.

For this project we were supposed to develop a client program that could send regular data as well as covert data. Then we were instructed to create a second program that would act as the server and receive the packet. This program needed to be able to not just interpret the normal data, but also the data sent covertly. For this experiment I decided to start with the IP Identification header manipulation and then move on from there.

For my test cases I wanted to first establish a baseline. I used 4 programs that were already created that I found online. One for each server and client of a TCP and UDP channel. Then I opened up WireShark and was able to capture packets being sent across the network. I monitored how a udp packet and tcp packet would look when they are implemented correctly without covert data. This would function as my correct implementation and I would know my covert channel was effective when my model looked very similar to this one. Then I was able to perform similar analysis on the tcp, udp packets. The things I focused on were anything that might be a red flag to a user. For example, if the whole packet got flagged as a malformed packet, if the checksum wasn’t computed correctly, if the TCP three way handshake didn’t look correct, etc. I also just looked through the different data that is displayed in wireshark to see if any of the covert data was readily available. If it was not

obvious then I knew I was on the right track. Unfortunately, things didn't go according to planned.

There were numerous problems I encountered during the implementation phase. The first step was just doing some more research into how tcp and udp packets are created. There were some great walk throughs out there that walk you through each field in a TCP/UDP/IP header and explain what it is for, when it is set, how often it is set, what type of data it contains, how big it is etc. After getting a little bit more of a background I into the finer details of the way these packets are formed I looked for a preexisting client server program. As mentioned in class, we did not have to generate our own basic client server program, but rather we could use one that was already created. This was the easy part. Through very basic google searches you can find all sorts of programs to send packets and receive packets in many different programming languages. I initially chose to go with a client and server application that was written in node.js. It was very clean looking, and only about 20 of lines of code altogether. However, both of these programs were dealing with normal socket libraries vs. raw socket libraries. This is where my first and biggest difficulty came in. I had to find an example program that utilized a raw-socket library that would allow me to send or receive tcp or udp packets. The reason I needed a raw socket was because I needed to be able to access all of these header fields in order to covertly add my data. This proved unreasonably difficult. I was able to find many, many examples that looked great, had walk-throughs, but then had a dozen errors when compiling. This led to my second problem.

The choice of which operating system would be best to work with for this was also difficult. I ended up switching back and forth between my Kali machine and my Mac OS hoping that I could find a program that would give me the desired results. After trying php, node, python, and C I finally settled on a client written in C. As far as I was concerned the server was less important at this point. After fixing all of the compile-time errors I tried to generate normal TCP packets using raw-sockets without adding any covert data. This led me to my next difficulty with implementation. I gauged whether or not the packet was being generated by looking at my server to see if it received the packet. After hours of trying to figure out why the packets weren't being generated I finally opened up Wireshark to see if maybe they were just being denied by the server for some reason. This was a huge break through. At this point I was able to see the packets being generated and I just needed to figure out why they weren't being accepted.

I thought maybe the checksum wasn't being calculated correctly so the packets were just being denied and dropped, so I enabled checksum validation in Wireshark. As expected, the checksum wasn't being generated correctly. Luckily, Wireshark calculate a checksum and tells you what your checksum should be. Using this I was able to put a hardcoded checksum into my packet to see if this would enable my server to receive it. Unfortunately it did not. I spent a few more hours trying to figure out why the server wouldn't accept it, and after not making much progress I moved on to trying to add covert data. I was able to manipulate the IP ID header to hold my covert data, even though the server couldn't receive it. I included an image of the packet captured from Wireshark. In theory I would send a little bit of data in each packet through this field, but I wanted to send more at once, so I converted a text message "ALOHA" into hexadecimal. Then I converted this hex into decimal. Then I

divided the number of numbers needed to make up my message by 4. In this case ALOHA is 12 numbers, and I took every 3 bits and placed it into the IP ID header. So the first number, the fourth number, the 7th number and the 10th number went into the IP ID field. Then I placed the rest of the numbers at the end of the payload. This way if someone was looking at the packets these bits on the end wouldn't make any sense. Even if they discovered the way the data was being encrypted, they would need to know to look at the IP ID field for what functions similarly to a key.

These channels can be very dangerous. The most likely use I can think of for them would be to get information out of a machine that an attacker has infiltrated. They would use a covert channel to exfiltrate the data. An attacker that has added some sort of malware to a machine that is supposed to leak information wants to get this information undetected. If an attacker can get information from a system undetected through a covert channel it is substantially more useful. For example, passwords that the user doesn't know are compromised won't be changed so an attacker could get access to a system until those passwords are reset by normal procedures. Covert channels can also be used to send information if encryption isn't available, but people still want to hide what they are saying. For example, if someone was trying to sneak information out of North Korea they could send UDP packets out with hidden data over their covert channel. These packets would not trigger an acknowledgement and would then be bypassing North Korea's security protocols.

This was the command to execute my client from the command line.

-Source IP (could spoof), source port (could spoof), dest IP, dest. port, number of attempts.

```
Tylers-MacBook-Pro:Desktop tylerjackson$ sudo ./newest -s 127.0.0.1 -d 127.0.0.1
 -p 6969 -n 10
```

This is what the packets looked like without raw sockets from Wireshark.

127.0.0.1	127.0.0.1	TCP	68 65203→1337 [SYN] Seq=0 Win=655
2605:6000:1508:c0a0:5e96:9(	2605:6000:1508:c0a0:9272:4(	ICMPv6	78 Neighbor Advertisement 2605:60
127.0.0.1	127.0.0.1	TCP	68 1337→65203 [SYN, ACK] Seq=0 Ac
127.0.0.1	127.0.0.1	TCP	56 65203→1337 [ACK] Seq=1 Ack=1 W
127.0.0.1	127.0.0.1	TCP	56 [TCP Window Update] 1337→65203
127.0.0.1	127.0.0.1	TCP	84 65203→1337 [PSH, ACK] Seq=1 Ac
127.0.0.1	127.0.0.1	TCP	56 1337→65203 [ACK] Seq=1 Ack=29
127.0.0.1	127.0.0.1	TCP	95 1337→65203 [PSH, ACK] Seq=1 Ac
127.0.0.1	127.0.0.1	TCP	56 65203→1337 [ACK] Seq=29 Ack=40
127.0.0.1	127.0.0.1	TCP	56 65203→1337 [FIN, ACK] Seq=29 A
127.0.0.1	127.0.0.1	TCP	56 1337→65203 [ACK] Seq=40 Ack=30
127.0.0.1	127.0.0.1	TCP	56 [TCP Dup ACK 11#1] 65203→1337
127.0.0.1	127.0.0.1	TCP	56 1337→65203 [FIN, ACK] Seq=40 A
127.0.0.1	127.0.0.1	TCP	56 65203→1337 [ACK] Seq=30 Ack=41

This is what the packets looked like that I created. Notice that I expanded the IP ID and the payload so that you can see how I added some numbers after the normal data “Hello” and the rest is in the IP ID.