Global Information Assurance Certification Paper

GIAC Certified Incident Handler (GCIH)

Practical Assignment

Version 2.1

Option 1 – Exploit in Action

"Paratrace Analysis and Defence"

By David Jenkins (GCIA)

Table of Contents

Introduction

This paper consists of an extensive review of the Paratrace reconnaissance tool in how it works and how it would be applied and the incident handling processes that should be followed in response to the use of that tool on a network. All testing was conducted on a test network for the research of the tool and for creating a hypothetical scenario for incident response process.

The Exploit

Name

The Paratrace exploit has been selected for detailed review. The tools were compiled and all testing conducted in the test network. Paratrace comes bundled in the Paketto download. There have been no CVE and CERT numbers assigned to this exploit. This is most likely because it is not a specific vulnerability that the tool target but utilises the way routers work on the Internet and therefore is not an actual coding error on the vendor's behalf, but a general weakness in the design of IPv4.

Operating System

The systems affected by this are any routing devices that comply with the IPv4 RFC's. The exploit itself runs a several flavours of Unix and Linux.

Protocols/Services/Applications

The protocols utilised in the exploit are TCP and ICMP.

Brief Description

Paratrace (Parasitic Trace) is a cleverly designed program that traces the route from the source to the destination, by 'piggy backing' on a current TCP connection. The output result is similar to Unix traceroute and Windows tracert, but works in a different manner. Paratrace utilises well selected time to live TCP messages for each router and collates the information received back in the time exceeded replies. Because of the different way it works, it is possible Paratrace is able to identify routing devices behind a stateful packet firewall which may have also been network address translated.

Variants

No variants known.

References

Following are relevant references to the program itself.

- Kaminsky, Dan, URL http://www.doxpara.com
- Kaminsky, Dan, "Paketto Brief" URL http://www.doxpara.com/read.php/docs/pk_english.html
- Kaminsky, Dan, "Paketto-1.1". Version 1.1 URL http://www.doxpara.com/paketto/paketto-1.1.tar.gz

- Kaminsky, Dan, “BH-Asia-02-kaminsky.pdf” URL http://www.blackhat.com/presentations/bh-asia-02/Kaminsky/bh-asia-02-kaminsky.pdf

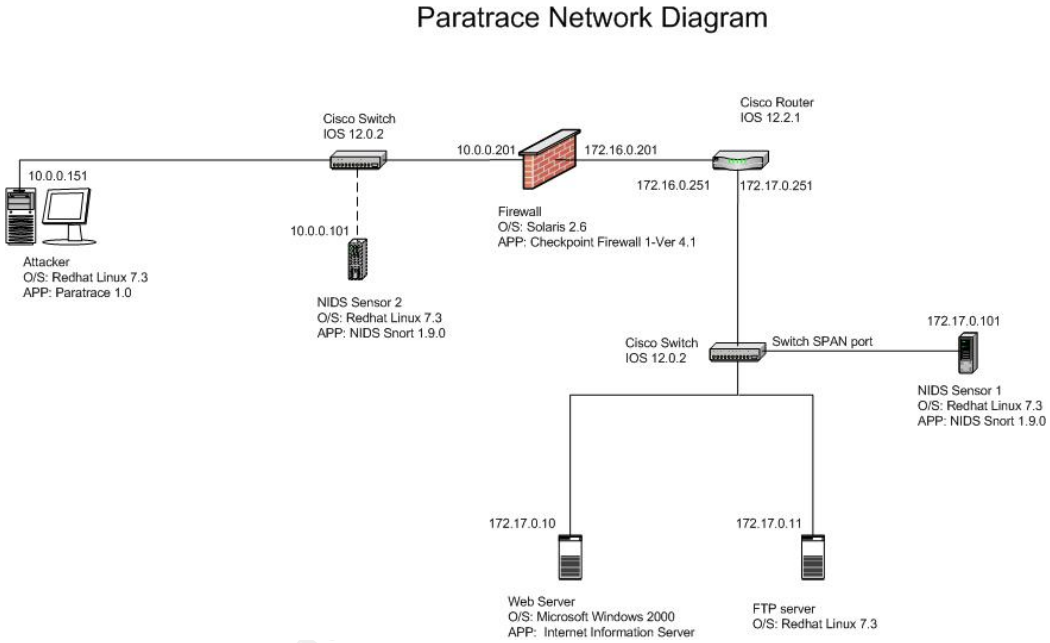
- DeJesus, Edmund X, “Paketto Keiretsu: New TCP/IP Tools Cut Both Ways “. URL http://www.infosecuritymag.com/2002/nov/digest25.shtml#news1

- Johnston, Gretel. “The Golden Age Rolls On”. URL http://www.computerworld.com/securitytopics/security/holes/story/0,10801,75381,00.html

The Attack

This section describes how the tool could be used to perform reconnaissance on a network. A test network was created and the tool run against it. For illustration purposes, port 80 HTTP is used to demonstrate the attack. This reconnaissance can just as easily work with the FTP service, or any other TCP service that is being hosted by the network.

Description and diagram of network



Protocol description

The exploit works by utilising weaknesses in IP protocols TCP, ICMP and the way stateful firewalls work.

In the TCP protocol if a packet is not acknowledged within a certain timeframe, then a duplicate packet is sent. So it is perfectly acceptable and normal for duplicate packets to be seen from the sender on TCP/IP networks. Below a section from the DOD standard on TCP which reflects this.

> DOD STANDARD - TRANSMISSION CONTROL PROTOCOL
>
> 2.6. Reliable Communication
>
> When the TCP transmits a segment, it puts a copy on a retransmission queue and starts a timer; when the acknowledgment for that data is received, the segment is deleted from the queue. If the acknowledgment is not received before the timer runs out, the segment is retransmitted (DOD Standard TCP, Section 2.6)

Although this adds the to robustness of TCP, it can be a double edged sword. These duplicate packets can be used by an attacker for means other than the intended purpose.

The ICMP (Internet Control Message Protocol) is used to communicate IP communication problems. RFC792 is the protocol specification for ICMP, and details all requirements for ICMP, including but not limited to error messages to the sender:

- if the packet cannot reach the destination,
- hop count exceeded for TTL (time exceeded),
- if destination port not available,
- echo requests...etc.

So although ICMP is essential in the proper functioning of IP, it can be used also by attackers for reconnaissance. Many network scanning tools utilise ICMP in some way. For example tools that scan for UDP ports, will try to connect to a particular port, say 1234, and if there is no program servicing that port, then an ICMP Type 3, Code 3 “Port Unreachable” will be sent back to the sender.

How the exploit works

Brief Description

1. Attacker runs Paratrace program with the target of the web server
2. Attacker then connects to web server
3. The Paratrace program creates duplicates of the TCP packets and sends them onto the target network. These packets have low TTL values.
4. Routing devices that see the TCP packets with TTL 1, decrement the value to 0, drop the packet and send an ICMP "Time Exceeded" message back to the originator of the TCP packets, the attacker.
5. Attacker receives the ICMP messages and creates a map of the internal network.

Detailed Description

The first step Paratrace does is watch packets that pass the network interface and filters out anything but ICMP packets or TCP packets with the target address in the IP header part of the packet. Paratrace utilises the pcap libraries for some of the packet handling routines.

This first part of the program is done in the following lines of code from paratrace.c source file, refer to Appendix 2 and 3 for the relevant source code modules. All source referenced in this paper is from the Paketto suit, the author being Dan Kaminsky.

snprintf(pfprogram, sizeof(pfprogram), "icmp or (src %s %s and tcp)", buf, target);

This source line copies the relevant packet search parameters (being ICMP, target host or net and TCP connections) into the string pfprogram.

if (pcap_compile(pcap, &fp, pfprogram, 1, 0x0) == -1) {

This source line compiles the search parameter string pfprogram into the format pcap requires for use, and is encapsulated in an error checking if statement.

if (pcap_setfilter(pcap, &fp) == -1) {

This source code line, actually sets the filter to the pcap library to only look at the required packets as described in the filter.

The program then goes into a while loop which has 3 main 'if' segments of code, these are

1) if(target_acquired && x.ip->ip_p == IPPROTO_ICMP)

2) else if(!up && target_acquired && x.ip->ip_p == IPPROTO_TCP)

3) else if(!target_acquired && x.ip->ip_p == IPPROTO_TCP && x.tcp->th_flags == TH_ACK)

As the variable target_aquired is set to 0 at the start of the program, then the first thing the program does, is wait for a TCP ACK packet. Once it find this packet the following happens

- The program estimates the hop count to the target

```
snprintf(ttlrange, 1024, "1-%u", estimate_hopcount(x.ip ->
ip_ttl)+hopfuzz);
```

- The program set's up the reply to the TCP ACK packet by switching the source destination host and port values

```
pk_memswp(&(x.eth->ether_dhost), &(x.eth ->
ether_shost),ETHER_ADDR_LEN);

pk_memswp(&(x.ip->ip_src), &(x.ip->ip_dst),
IPV4_ADDR_LEN);

pk_memswp(&(x.tcp->th_sport), &(x.tcp->th_dport), 2);

pk_memswp(&(x.tcp->th_seq), &(x.tcp->th_ack), 4);
```

- Zero's the payload by setting the packet length to end at the end of the packet header.

```
x.ip->ip_len = htons((int)x.ip->ip_hl*4 + (int)x.tcp->th_off*4);

pkthdr.caplen = LIBNET_ETH_H + (int)x.ip->ip_hl*4 + (int)x.tcp-
>th_off*4;
```

- Re-calculates the IP checksum.

```
recalc_checksums(&x, IPPROTO_TCP);
```

- Calls raw_sock_syn_scan from scanutil.c from paketto suite, passing ttlrange value.

```
raw_sock_syn_scan(dest, sizeof(dest), dev, &x,
        ttlrange, seed, bandwidth, verbose, resolve, 1);
```

  - Relevant parts from Raw_sock_syn_scan procedure for paratrace are that it loops as many times as there are hops to target

```
for(ttl=start_ttl; ttl<=end_ttl; ttl++){
```

- Sets ttl to start ttl value and puts in IP TTL field

scanx->ip->ip_ttl = ttl;

- Puts the TTL value into the IP ID field\

scanx->ip->ip_id = htons(ttl);

- Sends the packet

i = libnet_write_ip(sockfd, (char *)scanx->ip, (int)scanx->ip->ip_hl*4 + (int)scanx->tcp->th_off*4);

This is the stage where the program is utilising duplicate packet allowable state for TCP. Paratrace takes advantage of this and creates a duplicate packet of an outgoing legitimate TCP packet and inserts a low TTL (Time To Live) in the IP header. In the Paratrace Network Diagram above, the first packet would have a TTL of 1. The next packet sent would have a TTL of 2 and so on.

Once the first packet hits the first router, the TTL value will be decremented and will be dropped by the router because TTL is now 0. The router will also issue an "ICMP Time Exceeded" message back to the attacker. The following packets, once TTL has been reduced to 0 will also produce time exceeded messages.

Paratrace utilises the ICMP Type 11, Code 0 "Time exceeded" in transmit messages to map out the routing devices between the source and destination IP addresses.

The ICMP replies are captured in the first of the main ifs in paratrace.c

if(target_acquired && x.ip->ip_p == IPPROTO_ICMP)

In this section of code, it takes the ICMP Time Exceeded message and prints the relevant information from the packet. Sample output from http://www.doxpara.com/read.php/docs/paratrace.html

```
002 =   63.251.53.219|80    [02]   5.170s(      10.0.1.11 ->
66.35.250.150   )
001 =      64.81.64.1|80    [01]   5.171s(      10.0.1.11 ->
66.35.250.150   )
003 =    63.251.63.14|80    [03]   5.195s(      10.0.1.11 ->
66.35.250.150   )
  UP:    66.35.250.150:80    [12]   5.208s
004 =   63.211.143.17|80    [04]   5.219s(      10.0.1.11 ->
66.35.250.150   )
005 =   209.244.14.193|80    [05]   5.235s(      10.0.1.11 ->
66.35.250.150   )
```

```
006 =  208.172.147.201|80    [08]   5.273s(       10.0.1.11 ->
66.35.250.150   )
007 =  208.172.146.104|80    [06]   5.277s(       10.0.1.11 ->
66.35.250.150   )
008 =  208.172.156.157|80    [08]   5.314s(       10.0.1.11 ->
66.35.250.150   )
009 =  208.172.156.198|80    [08]   5.315s(       10.0.1.11 ->
66.35.250.150   )
010 =    66.35.194.196|80    [09]   5.337s(       10.0.1.11 ->
66.35.250.150   )
011 =    66.35.194.58|80     [09]   5.356s(       10.0.1.11 ->
66.35.250.150   )
012 =    66.35.212.174|80    [10]   5.379s(       10.0.1.11 ->
66.35.250.150   )
```

The crux of TCP legitimately having multiple packets is that Paratrace can go through stateful packet firewalls. This works in the following way. Referring to the "Paratrace Network Diagram" above, the firewall has a rule in it as below:

Source (ANY), Destination (Web Server), Service (HTTP), Action (Allow)

This rule allows any source IP address connect to the web server using HTTP. This is a very common rule to find in corporations so that their company web pages can be viewed by the public. The following steps occur:

1) Attacker using their web browser connects to the target web site.
2) Firewall checks packet against rules and allows the connection.
3) There is now a new entry in the firewall state table that says:

Attacker – Web server – HTTP – Active

4) Attacker now sends a duplicate TCP packet to the web server. Firewall checks to see if the packet is part of an already active TCP state, finds a match and allows the packet through. Although in this case, the duplicate packet has a TTL trigger to map the network.

The program runs on flavours of Unix and Linux and can be used in simple following manner.

paratrace www.sans.org

Then on the same machine, browse to www.sans.org. The program will then use the browsed connection to operate.

There are a number of command line options that can be used with paratrace. The list below are the available options with a description of what they do.

-s [hopfuzz]: This option allows the user to select the number of hops to go further than the program estimates the target is. This can ensure

that if the program does not calculate the hop distance correctly, then the whole path is followed. Selecting a value of –1 here reduces the number of duplicate packets sent, and can aid in the degree of stealth.

-t [timeout]: This is the time the program will wait for the response to the last packet before ending.

-b[bandwidth]: Allows the user to limit bandwidth consumption. Options of 0/b/k/m/g. Implemented by using usleep c command. b=64 seconds, k=6.2 milliseconds, m= 61 microseconds, 0.06 microseconds. If 0 selected no delays and maximum bandwidth used. Default is 0.

Relevant source code

- case 'b': multiple=1;
- case 'k': multiple=1024;
- case 'm': multiple=1024*1024;
- case 'g':multiple=1024*1024*1024;
- i=base*multiple;
- packetsleep=(1000000*64)/i; /*64 is min. frame size*/
- if(packetsleep)usleep(packetsleep);

-n : Allows the user to specify a target network instead of a target address. This has the effect of any TCP packets from a host on that network can be used for paratrace.

-N/-NN : Use –N reverse DNS lookups to determine the name of the source address. Use –NN to do so with destination address instead of source address.

-v : Go into verbose mode level 1. Display sending of packets.

-vv : Go into verbose mode level 2. This includes outputting all interpreted TCP and IP headers.

-d [device]: Use this device for all traffic.
-i [source]: Use this source IP address for all traffic.

Theoretically it would also be possible to achieve the same results as paratrace without the program, although it would be slow, and take more effort and timing issues may come into play. To do this, one would need the following:

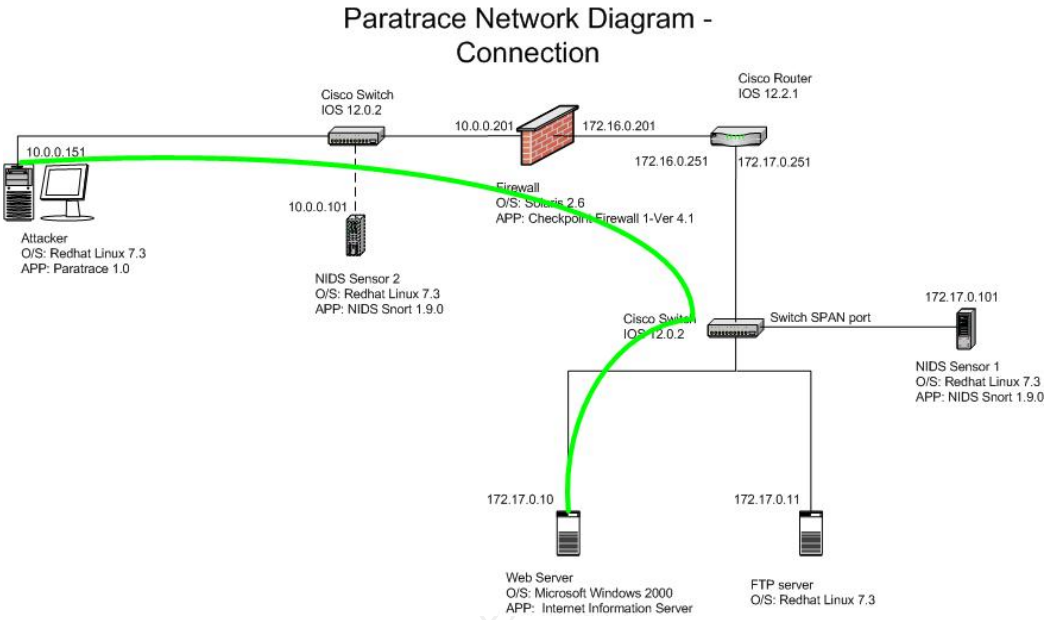
- Their favourite packet crafting tool of the month,
- A packet capture program, tcpdump would be sufficient.

First, the person would run tcpdump and capture the TCP ACK packet. Then take this packet and manipulate the information. i.e. – switch source destination addresses and port, zero the payload, put the required TTL value

in the IP header and recalculate the checksums then inject the packet onto the wire. To put the correct TTL value in, this would have to be worked out by the person. Then repeat this process as many times as there are hops between the themselves and the target. This would be a slow and tedious process and naturally that is what computer programs are for.

Description and diagram of the attack

1) The attacker enters the command paratrace www.targetwebserver.com
2) They then browse to the target web server.



Tcpdump command is used to shows all ICMP packets or packets with port 80 as the source or destination port. The –v option shows output in verbose mode, in particular we are interested in the TTL values.

tcpdump -v -n ip[9] = 1 or port 80

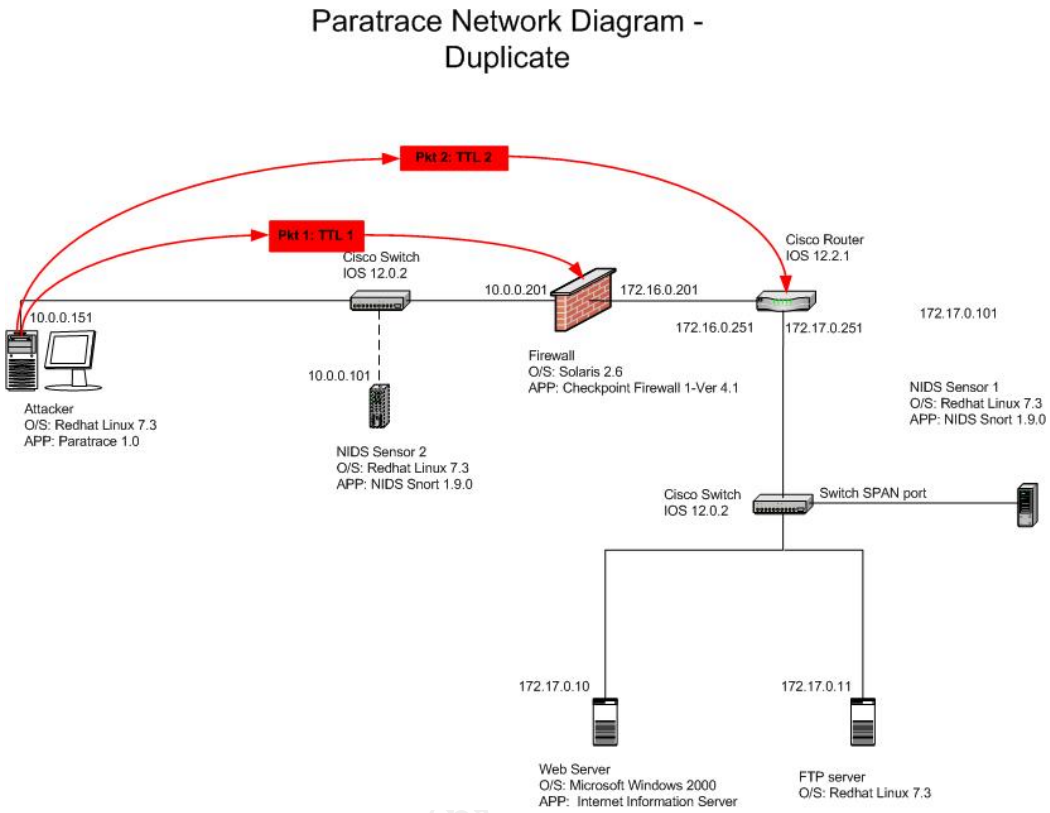
20:43:12.500095 10.0.0.151.1084 > 172.17.0.10.80: S 2115551386:2115551386(0) win 5840 (DF) (ttl 64, id 50290, len 60)
20:43:12.503967 172.17.0.10.80 > 10.0.0.151.1084: S 3623596701:3623596701(0) ack 2115551387 win 10136 (DF) (ttl 247, id 21697, len 60)
20:43:12.504035 10.0.0.151.1084 > 172.17.0.10.80: . ack 1 win 5840 (DF) (ttl 64, id 50291, len 52)
20:43:12.521246 172.17.0.10.80 > 10.0.0.151.1084: P 1:43(42) ack 1 win 10136 (DF) (ttl 247, id 21698, len 94)
20:43:12.523618 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 5840 (DF) (ttl 64, id 50292, len 52)
20:43:15.998353 10.0.0.151.1084 > 172.17.0.10.80: P 1:9(8) ack 43 win 5840 (DF) (ttl 64, id 50293, len 60)

20:43:16.001112 172.17.0.10.80 > 10.0.0.151.1084: . ack 9 win 10136 (DF) (ttl 247, id 21699, len 52)
20:43:16.001674 172.17.0.10.80 > 10.0.0.151.1084: P 43:73(30) ack 9 win 10136 (DF) (ttl 247, id 21700, len 82)
20:43:16.002595 10.0.0.151.1084 > 172.17.0.10.80: . ack 73 win 5840 (DF) (ttl 64, id 50294, len 52)
20:43:16.031099 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136 (DF) [ttl 1] (id 1, len 52)
20:43:16.031497 10.0.0.201 > 10.0.0.151: icmp: time exceeded in-transit (ttl 255, id 50784, len 56)

20:43:16.044021 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136 (DF) (ttl 2, id 2, len 52)
20:43:16.044686 150.122.58.249 > 10.0.0.151: icmp: time exceeded in-transit (ttl 254, id 34338, len 56)

The trace in green is the http connection handshake and data packets from the browser on the attackers machines and the target web server.

3) Paratrace, detects the TCP packets and sends out the duplicates with ascending TTL values.



Below is the tcpdump output again:

20:43:12.500095 10.0.0.151.1084 > 172.17.0.10.80: S 2115551386:2115551386(0) win 5840 (DF) (ttl 64, id 50290, len 60)

20:43:12.503967 172.17.0.10.80 > 10.0.0.151.1084: S 3623596701:3623596701(0) ack 2115551387 win 10136 (DF) (ttl 247, id 21697, len 60)

20:43:12.504035 10.0.0.151.1084 > 172.17.0.10.80: . ack 1 win 5840 (DF) (ttl 64, id 50291, len 52)

20:43:12.521246 172.17.0.10.80 > 10.0.0.151.1084: P 1:43(42) ack 1 win 10136 (DF) (ttl 247, id 21698, len 94)

20:43:12.523618 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 5840 (DF) (ttl 64, id 50292, len 52)

20:43:15.998353 10.0.0.151.1084 > 172.17.0.10.80: P 1:9(8) ack 43 win 5840 (DF) (ttl 64, id 50293, len 60)

20:43:16.001112 172.17.0.10.80 > 10.0.0.151.1084: . ack 9 win 10136 (DF) (ttl 247, id 21699, len 52)

20:43:16.001674 172.17.0.10.80 > 10.0.0.151.1084: P 43:73(30) ack 9 win 10136 (DF) (ttl 247, id 21700, len 82)

20:43:16.002595 10.0.0.151.1084 > 172.17.0.10.80: . ack 73 win 5840 (DF) (ttl 64, id 50294, len 52)

20:43:16.031099 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136 (DF) [ttl 1] (id 1, len 52)

20:43:16.031497 10.0.0.201 > 10.0.0.151: icmp: time exceeded in-transit (ttl 255, id 50784, len 56)

20:43:16.044021 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136 (DF) (ttl 2, id 2, len 52)

20:43:16.044686 150.122.58.249 > 10.0.0.151: icmp: time exceeded in-transit (ttl 254, id 34338, len 56)

The lines in red are the actual trace of the packets with low TTL's sent by paratrace. The first red trace packet, shows a TTL of 1 and an id of 1. The second red trace packet shows a TTL of 2 and an id of 2 as per the diagram. This is concurrent with the source code:

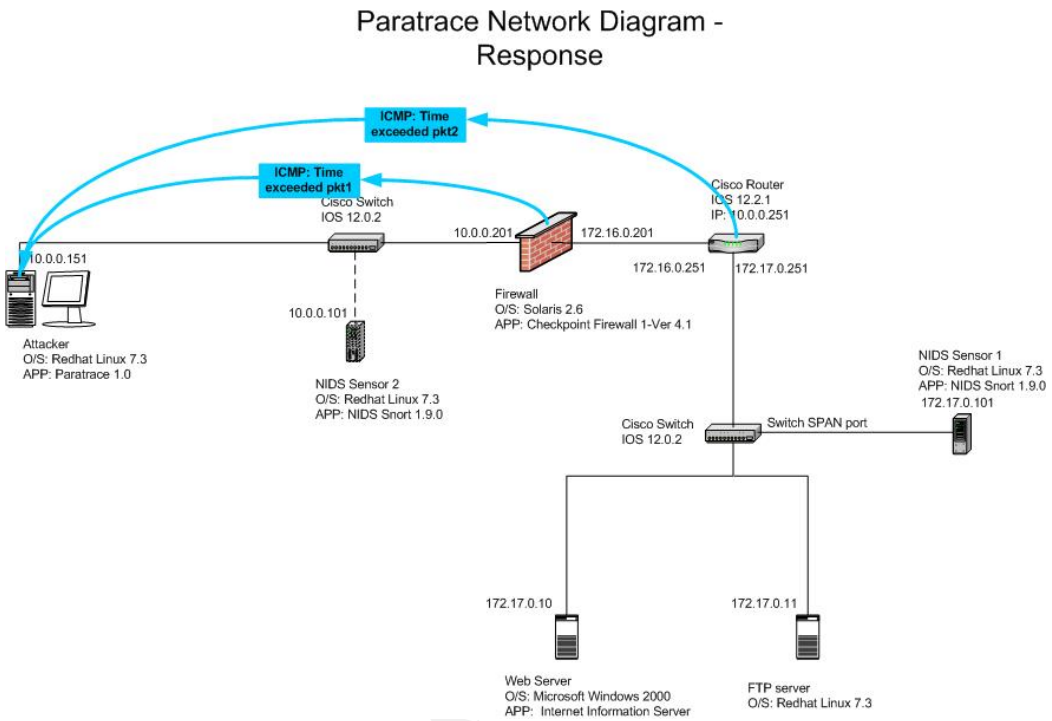
- Sets ttl to start ttl value and puts in IP TTL field

```
scanx->ip->ip_ttl = ttl;
```

- Puts the TTL value into the IP ID field\

```
scanx->ip->ip_id = htons(ttl);
```

4) Paratrace collates the ICMP time exceeded replies.



Below is the tcpdump output again:

20:43:12.500095 10.0.0.151.1084 > 172.17.0.10.80: S 2115551386:2115551386(0) win 5840 (DF) (ttl 64, id 50290, len 60)

20:43:12.503967 172.17.0.10.80 > 10.0.0.151.1084: S 3623596701:3623596701(0) ack 2115551387 win 10136 (DF) (ttl 247, id 21697, len 60)

20:43:12.504035 10.0.0.151.1084 > 172.17.0.10.80: . ack 1 win 5840 (DF) (ttl 64, id 50291, len 52)

20:43:12.521246 172.17.0.10.80 > 10.0.0.151.1084: P 1:43(42) ack 1 win 10136 (DF) (ttl 247, id 21698, len 94)

20:43:12.523618 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 5840 (DF) (ttl 64, id 50292, len 52)

20:43:15.998353 10.0.0.151.1084 > 172.17.0.10.80: P 1:9(8) ack 43 win 5840 (DF) (ttl 64, id 50293, len 60)

20:43:16.001112 172.17.0.10.80 > 10.0.0.151.1084: . ack 9 win 10136 (DF) (ttl 247, id 21699, len 52)

20:43:16.001674 172.17.0.10.80 > 10.0.0.151.1084: P 43:73(30) ack 9 win 10136 (DF) (ttl 247, id 21700, len 82)

20:43:16.002595 10.0.0.151.1084 > 172.17.0.10.80: . ack 73 win 5840 (DF) (ttl 64, id 50294, len 52)

20:43:16.031099 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136 (DF) [ttl 1] (id 1, len 52)

20:43:16.031497 10.0.0.201 > 10.0.0.151: icmp: time exceeded in-transit (ttl 255, id 50784, len 56)

20:43:16.044021 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136 (DF) (ttl 2, id 2, len 52)

20:43:16.044686 150.122.58.249 > 10.0.0.151: icmp: time exceeded in-transit (ttl 254, id 34338, len 56)

The trace packets in blue are the time exceeded messages back from the routers, in this case the firewall is acting as a router and so sends the ICMP message as well.

The resultant output from paratrace is as below.

Paratrace 172.17.0.10

Waiting to detect attachable TCP connection to host/net: 172.17.0.10
172.17.0.10:80/32 1-15
001 = 10.0.0.201|80 [01] 2.362s(10.0.0.152 -> 172.17.0.10)
002 = 172.16.0.251|80 [02] 2.378s(10.0.0.151 -> 172.17.0.10)
UP: 172.17.0.10:80 [03] 3.319s

Signature of the attack

The initial test network did not detect the attack. This was because the NIDS system was placed on the network just before the web and ftp server, and the duplicate probing packets did not go that far.

The NIDS was then moved and placed in front of the firewall as per the dotted line in the diagram and the paratrace program run again.

The following snort alert was logged

```
[**] ICMP Time-To-Live Exceeded in Transit [**]
12/25-20:43:13.669085 10.0.0.201 -> 10.0.0.151
ICMP TTL:255 TOS:0x0 ID:50784 IpLen:20 DgmLen:56
Type:11  Code:0  TTL EXCEEDED IN TRANSIT
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=
+=+=+=+=+=+=+=+
[**] ICMP Time-To-Live Exceeded in Transit [**]
12/25-20:43:13.680231 172.16.0.251 -> 10.0.0.151
ICMP TTL:254 TOS:0x0 ID:34338 IpLen:20 DgmLen:56
Type:11  Code:0  TTL EXCEEDED IN TRANSIT
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=
+=+=+=+=+=+=+=+
```

This was using the latest version of Snort at the time, version 1.9.0 and the latest ruleset.

The specific signature that matched was from the icmp-info.rules file and was:

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP Time-To-Live Exceeded in Transit"; itype: 11; icode: 0; sid:449;  classtype:misc-activity; rev:4;)
```

The problem with this detect, is that it will match on any valid ICMP timeout messages and not only alert when paratrace is being used. In the IDS field, much time is spent on trying to minimise the number of false positives, so this would be no good.

A very simple way of detecting the paratrace program would be to look for any TCP packets sent out that had a TTL value matching the IP ID field. Here are the two duplicate packets from the trace above that were sent by paratrace. The parts in red are the TTL and IP ID field. We can see that they are equal.

```
20:43:16.031099 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136  (DF)
(ttl 1, id 1, len 52)
20:43:16.044021 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136 (DF)
(ttl 2, id 2, len 52)
```

The detect can be accomplished by using the following tcpdump command:

tcpdump –v ip[9] = 6 and ip[8:1] == ip[5:1]

Which translates to:show packets of TCP protocol where TTL = IP ID

There were no other traces found in the firewall logs or on the router. To have this information logged a specific rule would need to be added into the firewall ruleset and configured to log the icmp packets. Also on the router the logging would need to be turned on, although this can quickly fill up the available space on the router.

How to protect against it

To stop paratrace being able to map out the internal network a number of measures can be taken to control the traffic. Firstly we need to understand how the firewall controls which packets can and which packets cannot pass through the firewall.

The usual method on setting up the ruleset on checkpoint firewall-1 is to have a list of rules which permit traffic that is required and the last rule drops everything else. Drop everything unless specifically granted.

Test Network Firewall Ruleset

- Rule 1 – Permit any source IP destined for WebServer using protocol HTTP
- Rule 2 – Permit any source IP destined for FTPServer using protocol FTP
- Rule 3 – Drop any external source IP's destined for the internal network using protocol ICMP Type Echo Request
- Rule 4 – Permit Firewall Manager destined for Firewall using FW1 protocol
- Rule Last – Drop any source IP destined for any destination IP any protocols.

A packet coming into the firewall, would first be checked against Rule 1, to see if it is permitted. If it is, the packet is passed through and there is no more checking against the other rules. If it is not specifically permitted by the first firewall rule then it moves onto the next one and checks the packet against that. If it is permitted is passes the packet through and moves onto the next packet. If a packet is checked against all the rules and has still not been permitted, it reaches the last rule, which is drop everything. This of course matches and so the packet is dropped.

One may well ask why in the test network, the ICMP messages were actually passed through the firewall without any specific rules stating that ICMP is allowed through. This is because apart from the ruleset checked against, Checkpoint Firewall-1 has some extra rules which are not seen here. There is a rule 0 which always comes first which handles a number of checks including anti-spoofing. Additionally, in the security policy of the firewall there are a

number of options that can be selected. One of these is “Allow ICMP” messages. If this is selected, then an automatic rule is added before the last one, that allows ICMP messages. This was selected in the test network firewall.

To stop the ICMP time exceeded messages from exciting back through the firewall, one of two steps can be taken. De-select the “Allow ICMP” field in the security policy of the firewall, which because there is no specific rule allowing ICMP, would then drop all ICMP by the final rule. This will have the added effect of also stopping all ICMP messages coming into the internal network, which may cause problems because of the possibility of blocking legitimate ICMP messages.

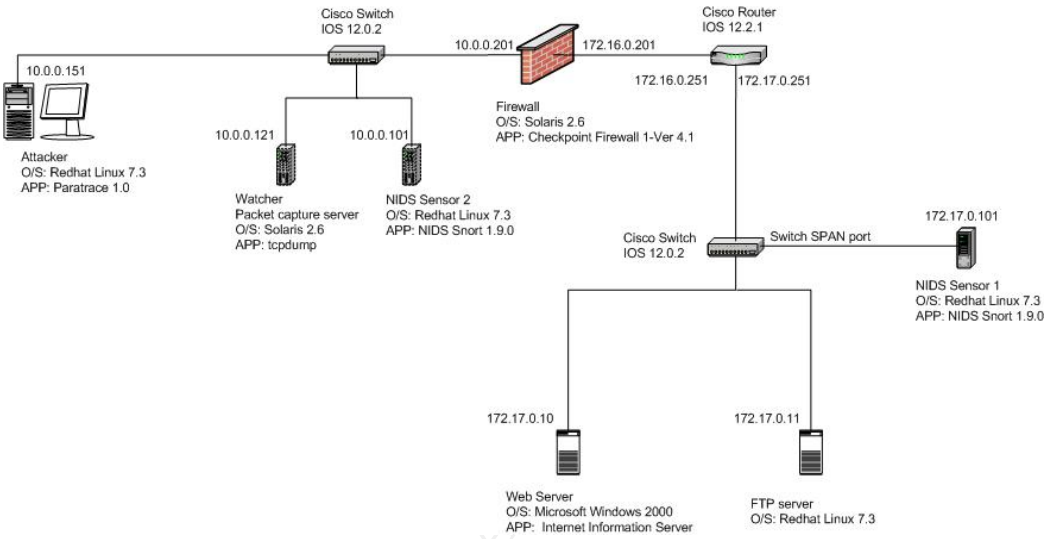
The other options is if you still want ICMP messages but only stop the ICMP “Time Exceeded” messages, de-select the “Allow ICMP” field and put a specific rule in the firewall, which drops packets of ICMP Type 11 Code 0. Turning the logging on for this rule would be a good way to see how many attempts are being made to try and map your network using expired TTL’s. Although there may be some legitimate time exceeded messages, most of them will be the result of scanning attempts. This second method is more secure, as you need to specifically state exactly what ICMP is allowed through the firewall, instead of allowing a carte blanche effect for all ICMP. Although, this will take more effort in determining exactly what ICMP messages you want to allow to pass in and out of your network.

From the vendor perspective, there is not too much that can be done as this is mainly a design issue of IP version four. Checkpoint should make sure that the “Allow ICMP” field in the security policies of the Firewall-1 product of all version and all new versions is not selected by default and a better explanation in the help section of there product describing the risks associated with selecting this option. The router vendors, could put some logic in their products that checked all packets to see if the TTL equalled the IP ID value and drop the packet if it did. Although their stance on this may be that their devices are just focusing on routing and not providing security checks. An option for them would be to include this is a security plug-in for their routers which would be optional for the owner to utilise or not.

The Incident Handling Process

This section describes what incident handling process should be followed to address the attack described above on the “Company Network Diagram” below.

Company Network Diagram



Preparation

Prior to any incident, there should already be counter measures, an incident handling process and policy statements and an incident handling team already setup so as when an incident does happen the company is prepared to deal with it.

Counter Measures

There should be a number of counter measures in place ready to be applied. This can range from IDS, disconnecting the affected system from the network, powering down the server, applying blocking rules in the firewall and applying router controls.

In this case the counter measure was the IDS system which detected the paratrace attempt in the first place.

Incident Handling Team

There should be a well defined incident handling team. This should consist of

The incident coordinator – This person is responsible for coordinating the whole incident. They will call in and direct each of the team and actions they will take.

Security – A member from the security group to be able to provide subject matter technical expertise in regards to the incident.

System support/administration – This will be a minimum select number of people, depending on how the support structure works in the company. For example, there may be a person from the Unix support team, the Microsoft support team, the Mainframe support team and the Network support team. These people will need to be hand picked, well trusted and have worked for the company for an extended time.

Localised technical support – If the company is geographically disperse, then the incident coordinator will need to call on the localised support person in relation to the incident. If the company has many different locations, then the total number of localised people involved in the incident response team may be several tens of people. For each incident though, only the relevant localised technical people to deal with the incident should be contacted and involved, not the whole group. This complies with the incident handling best practice of the less people that know about the incident the better.

Legal – The legal team member, will need to be able to advise on legal matters as they arise. Also, they may need to prepare certain legal documents.

Human resources – This member will advise on staff matters, how to handle communications to the staff if necessary and what actions in relation to the staff member may need to be taken.

Public relations – This team member will handle all communication with the public. If any people external to the incident handling team ask questions in relation to the incident then they should be directed to the public relations person. This leads to the need to supply the public relations member enough information about the incident so they can appropriately inform people as per their procedures.

Excerpts from policy and procedure

Following are excerpts from the policy that support the incident handling and ensure that the company is prepared to respond to an incident.

- You shall report all incidents and potential incidents that come to your notice to your local help desk.
- The security group will investigate all security incidents and suspected incidents
- Security event logging shall be implemented on all systems.
- Security events logs must be protected against unauthorised access or modification

- Security event logs must be retained for a period not less than three months
- All security events must be centrally monitored
- All networked devices that support time synchronisation must synchronise to a single time source
- Where it appears that a breach has occurred, access to or use of the affected systems should be minimised until further notification.
- In the vent of an incident an Incident Coordinator shall be nominated.
- At the discretion of the security group details of an incident may be removed and replaced with a reference to a security investigation file.
- All media enquiries in relation to an incident shall be referred to the media relations group.
- If warranted the incident coordinator may involve other parties such as the legal group, human resources and media relations.
- Only those person directly involved in the incident handling and notification process may be informed of the existence or details of the incident.
- Reports detailing the nature volume and cost of incidents must be supplied to management on a monthly basis.
- Upon resolution of all the serious incidents a review must be conducted in order to identify the root cause and prevent recurrence.
- Incident response procedures must be tested on a annual basis.

Identification

Detection and Confirmation of Incident

The incident would be detected by the front network intrusion detection sensor alarming on the ICMP "Time Exceeded" messages. To confirm it was an incident a number of steps would have to be taken (evidence procedures are not described in this section, refer chain of custody section below for the procedures).

1) First, look at how many and how far apart in time the ICMP messages are coming. The number and close spaced frequency would be a sign of first suspicion of non-standard traffic.
2) Investigate the snort logs and determine where the reply ICMP messages are being sent.
3) The tcpdump server out the front of the firewall is performing a packet capture of all packets going to the firewall. This is captured in a file, which because of it's size would be rotated on a regular basis, a 24 hour time period per rotation is a good start point. This will depend on the amount of traffic passing the sensor and the storage capacity of the server. It would be necessary to perform a tcpdump on the current capture file for all packets from the attacker to see if there is anything strange in the packets or not. Appropriate command line would be:

   tcpdump –v <attackerIP> -r <filename>

From close inspection we would see the suspicious values of TTL and IP ID being the same (highlighted in red).

20:43:16.031099 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136 (DF) (ttl 1, id 1, len 52)
20:43:16.044021 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136 (DF) (ttl 2, id 2, len 52)

4) At this stage, the incident would be confirmed as a security incident.

Counter measures of incident

The counter measure used in this case was to detect the reconnaissance coming into the network as it happened using the snort IDS system situated out the front of the firewall.

Incident Details

The initial incident would be reported fairly quickly by the front IDS system. The log files of snort would show the following

Snort Log:
[**] ICMP Time-To-Live Exceeded in Transit [**]

12/25-20:43:13.669085 10.0.0.201 -> 10.0.0.151
ICMP TTL:255 TOS:0x0 ID:50784 IpLen:20 DgmLen:56
Type:11 Code:0 TTL EXCEEDED IN TRANSIT
=+=
+=+=+=+=+=+=+=+
[**] ICMP Time-To-Live Exceeded in Transit [**]
12/25-20:43:13.680231 172.16.0.251 -> 10.0.0.151
ICMP TTL:254 TOS:0x0 ID:34338 IpLen:20 DgmLen:56
Type:11 Code:0 TTL EXCEEDED IN TRANSIT
=+=
+=+=+=+=+=+=+=+

Confirmation would come from the tcpdump step as described above.
Tcpdump:

Investigation of the incident would take more time than the initial alert being acknowledged but would be fairly brief in this case. If the person is on site already, then the confirmation of the incident would be within the hour.

Chain of custody

An appropriate chain of custody process is:

The member of the responding team responsible for recording should:

- Complete an affirmation declaration
- Start a new Evidence Record form for each piece of evidence
- Assign an Evidence Number to the evidence
- Note the date and time that the evidence was found
- Describe the evidence
- Note the location in which the evidence was found
- Note who found the evidence
- Sign and date the form

The evidence should be placed in a plastic bag and retained by the member of the responding team responsible for recording. The completed Evidence Record should be attached to the bag.

The member of the responding team responsible for recording should maintain possession of the bag until such time as it can be placed in a secure room for analysis, or a safe for storage. The number of people who have access to this safe should be minimal.

Each time the evidence is moved from one place to the another the following details should be recorded

- Transferred from, By, Signature, Date
- Transferred to, By, Signature, Date
- Reason of transfer

The chain of custody part of the form is shown below. This section should be repeated at least as many times as the evidence is likely to change hands, so that the complete chain of custody can be recorded on the one form.

Chain of Custody			
Transferred From:		By:	
Signature:		Date:	
Transferred To:		By:	
Signature:		Date:	
Reason:			
Transferred From:		By:	
Signature:		Date:	
Transferred To:		By:	
Signature:		Date:	
Reason:			

Following is a flowchart of the evidence handling process.

For the attack above an appropriate affirmation would be:

"I, David Jenkins, 25th December 2002, am in Computer room 2A, 112 Spencer St Melbourne, and am looking at a Sun Sparc Ultra 30, serial number 122884EF3. This computer is suspected of containing evidence of criminal activity. At 21:14 I am removing one of the mirrored disks for evidence."

Listing of evidence

Mirrored hard drive from packet capture server, server name watcher, IP address 10.0.0.121

Containment

Assessment Process

The following assessment questions need to be answered in relation to the incident.

1. How widely deployed is the affected platform or application?
2. What is the effect of the exploitation?
3. Can the vulnerability be exploited remotely?
4. Is a public exploit available for the vulnerability and how easily is it obtained?
5. What level of skill and prerequisites are required by an attacker to exploit the vulnerability?
6. Is the vulnerability present in a default configuration?
7. Is a fix available for the vulnerability?
8. Do other factors exist which reduce or increase the vulnerability's risk or potential impact such as possibility it is a worm?

Below are the answers to the above questions in relation to the paratrace tool.

1. This exploit affects all routing devices.
2. The result of utilising the exploit is reconnaissance information
3. The vulnerability can be exploited anywhere on the internet.
4. Yes, the exploit is readily downloadable from http://www.doxpara.com/read.php/code/paketto.html
5. Skill level required to run this exploit is very minimal. Although an understanding of how best to use and which systems to target along with the meaning of the results requires some degree of network knowledge.
6. Not only the default behaviour, but the internet RFC's mandate that routers respond to the exploit.
7. No fix is available for the vulnerability, although a number of work arounds can be employed to stop the exploit be successful.
8. Exploit can only be successful if ICMP time exceeded replies messages are allowed through the firewall. Secondly the reconnaissance can only be able to map as far into the network as the deepest public accessible TCP service the company provides. This is supporting argument for having all public accessible services in the outer DMZ segment. Having a publicly accessible TCP service further in on the network provides the possibility of deeper reconnaissance into the internal network using paratrace.

Based on the answers above, this exploit should be addressed immediately.

Containment Process

As mentioned above, the first step of the process would be to pull out the mirror drive of the watcher server. This drive contains an exact copy of the main drive and therefore there is no need to make a backup. This makes the

process extremely simple and time efficient. Once the initial investigation has completed, then a new hard drive can be re-inserted into the watcher server and the create mirror command initiated to return the server back to the mirrored state.

The next step is to query and search through the current capture file on and locate the packets from the suspected source.

This would be done with the tcpdump command:

tcpdump –v <attackerIP> -r <filename>

20:43:12.500095 10.0.0.151.1084 > 172.17.0.10.80: S 2115551386:2115551386(0) win 5840 (DF) (ttl 64, id 50290, len 60)
20:43:12.503967 172.17.0.10.80 > 10.0.0.151.1084: S 3623596701:3623596701(0) ack 2115551387 win 10136 (DF) (ttl 247, id 21697, len 60)
20:43:12.504035 10.0.0.151.1084 > 172.17.0.10.80: . ack 1 win 5840 (DF) (ttl 64, id 50291, len 52)
20:43:12.521246 172.17.0.10.80 > 10.0.0.151.1084: P 1:43(42) ack 1 win 10136 (DF) (ttl 247, id 21698, len 94)
20:43:12.523618 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 5840 (DF) (ttl 64, id 50292, len 52)
20:43:15.998353 10.0.0.151.1084 > 172.17.0.10.80: P 1:9(8) ack 43 win 5840 (DF) (ttl 64, id 50293, len 60)
20:43:16.001112 172.17.0.10.80 > 10.0.0.151.1084: . ack 9 win 10136 (DF) (ttl 247, id 21699, len 52)
20:43:16.001674 172.17.0.10.80 > 10.0.0.151.1084: P 43:73(30) ack 9 win 10136 (DF) (ttl 247, id 21700, len 82)
20:43:16.002595 10.0.0.151.1084 > 172.17.0.10.80: . ack 73 win 5840 (DF) (ttl 64, id 50294, len 52)
20:43:16.031099 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136 (DF) [**ttl 1, id 1**, len 52)
20:43:16.031497 10.0.0.201 > 10.0.0.151: icmp: time exceeded in-transit (ttl 255, id 50784, len 56)
20:43:16.044021 10.0.0.151.1084 > 172.17.0.10.80: . ack 43 win 10136 (DF) (**ttl 2, id 2**, len 52)
20:43:16.044686 150.122.58.249 > 10.0.0.151: icmp: time exceeded in-transit (ttl 254, id 34338, len 56)

From looking at the packets above, it would be obvious that the red packets are the stimulus (specifically the bolded red part, being low TTL values) and the blue ICMP “Time Exceeded” messages are the response. The routers are sending ICMP messages back to the attacker and thus revealing their own IP addresses so the attacker can map the internal network. Successful reconnaissance is in action.

The next step would be to determine if the attacker has performed other work than just reconnaissance or has gone further, i.e. actually trying to penetrate systems. This can be done either looking through the firewall logs or

investigating the rest of the packets that were output from the tcpdump command above. At this stage it would be more thorough using the tcpdump command output as if there is some packets that require more detail, then the –X option prints out the captured part of the packet in hex and ASCII for detailed investigation.

The first three packets from the tcpdump are the TCP handshake and the next six packets are the http data transfer. This does not look suspicious and the important fact here is that there is only a total of nine packets that are not related to the paratrace reconnaissance. Secondly, the target is the same in each and that there is no other trigger alerts from the snort system to indicate any attempts to exploit known vulnerabilities.

The attacker could have after their scan, disconnected from their ISP and reconnected and thus have a new IP address. So it would be a good measure at this stage to check to make sure there are no IDS alerts recently, especially after the "ICMP Time Exceeded" ones.

Finding no further alerts it would be a fair decision to say that only reconnaissance has taken place at this stage. A further step would be to check if any other similar reconnaissance has occurred of the same type. This could be achieved with the following command line
tcpdump –v ip[9] = 6 and ip[8:1] == ip[5:1] –r <filename>

This would give a picture of how wide spread the reconnaissance is. In this case there was only the one attempt.

The appropriate containment measure to take would be to either stop the stimulus and or stop the response. As TCP packets for web traffic is allowed through the firewall and is needed for the operation of the web page, then the only simple control at this time would be to stop the response. Although the stimulus would still get through to the internal network the response would not and thus stop the mapping process from being successful. Secondly, the stimulus is not harmful to the routing devices. To do this it can be done by inserting a new firewall rule before the last rule which stops the ICMP messages. An appropriate rule would be:

Drop any "ICMP Time Exceeded" messages destined for outside the firewall.

As these ICMP messages are a normal part of the everyday internet, this could cause problem with using some of the companies internet services. Although with default values for TTL's at 64, 128 and 255, it is not frequent at all to see ICMP "Time Exceeded" messages unless a network has been setup incorrectly, possibly with a network loop somewhere.

Other containment measures, like changing root and administrative accounts passwords would not be necessary as there were no systems compromised.

Eradication

The first step of the eradication phase is to determine the cause and symptoms of the incident. At this stage we would know the following symptoms:

- The tools is a reconnaissance tool
- It uses low TTL values with matching IP ID values
- It triggers ICMP "Time Exceeded" messages from routing devices.
- It can go through stateful firewalls.

To try and determine the cause of the symptoms, a method would be to search for these systems with a good internet search engine. The following was typed into www.google.com

"mapping ttl id icmp time exceeded firewall stateful"

Each link would be investigated and found that the 6th link was a match to a presentation that the author of the reconnaissance tool gave at Hivercon in Dublin. The link being

www.doxpara.com/Black_Ops_Hivercon.ppt .

The searcher should now go to the base section of the URL and they would find available for download the paketto bundle which in the description on the web site describes a tool paratrace, that produces the symptoms found above. The tool should be downloaded and tested on a separate test network to see if it was actually the tool used for the reconnaissance. In this case it would be confirmed that paratrace was indeed the tool that was used. The options of the tool should be investigated and determined if after the containment controls were put in place, can the network still be vulnerable running the tool with different options. Additionally with this tool the source code is available. It could be determined fairly easily that there is no malicious code in the tool and that is would be safe to run the tool against the live network. This would show you the point of the view from the attackers perspective and what information is available. As in this case, all options of the tool depends on the ICMP "Time Exceeded" messages and so there is no further threat.

The next step of eradication is to improve the defences. In this cause it is known that the attacker has gained the IP address of the external interface of the firewall and the IP address of the router directly behind the firewall. There is a possibility of changing the internal IP address scheme so that the information that the attacker has gleaned is not valid anymore. Although this can required substantial effort, the risk and cost need to be weighed to justify this course of action.

At this stage it is prudent to perform a vulnerability analysis of the systems and network. This should be done on the systems and also from the network

vulnerability side and can be performed in house with available tools or a security consultant can be brought in to perform the work.

The actionable step in this section is to report the incident to the local CIRT and provide sanitised logs and traces in relation to the incident.

As no systems were actually compromised, there would be no need to follow the standard procedure of locating the most recent clean backup.

Recovery

The standard steps of recovery, decide when to restore, restore and validate the system would not be required to be actioned in this case as no systems were actually found to be compromised.

At this phase further investigation to determine if the paratrace traffic could be blocked before entering the internal network should be investigated. Or can the routing devices be configured to stop responding to the paratrace stimulus. This second option is doubtful as the ICMP "Time Exceeded" message is the normal operation of the routing device.

Trying to stop the packet entering the network in the first place cannot be done by the firewall. This is because the decision parameters of the firewall of whether to pass a packet or not, does not include looking at the detail of the TTL or the IP ID, and they cannot be compared to determine if these values are the same. As the firewall cannot block the traffic it should be investigated if the router that connects the company to the ISP can perform the blocking.

In the test network diagram there is no router in front of the firewall, but in a real situation there would be. Certain routers and models provide some functionality to be able to drop packets based on certain rules. Depending on what hardware is being run at the company, it may be possible to block packets that have a TTL value that matches the IP ID value. If this is the case, then the paratrace stimulus packet would not pass into the internal network in the first place. The problem with this control is that, the paratrace code can easily be changed so that the values of the IP ID field do not match the TTL value. It would be a fairly simple programming exercise to encrypt the IP ID value so the field would appear random.

A vital part of the recovery phase is to keep monitoring the network and see if the attack is continuing. If there is a continuous use from any IP address in relation to this type of traffic, then this may be a sign that the firewall control put in place is not working. This may be because the rule was not implemented correctly, or there is a variant of the reconnaissance tool that has been released that works differently, or simply just a new version of the tool which operates in a different manner.

Finally to ensure the exposure had been totally removed and after ensuring the code is not malicious in the eradication phase the tool should be run against the network and made sure that not internal information is gained from running it. In this case, as the ICMP "Time Exceeded" messages are blocked at the firewall, then no reconnaissance information would be gained by using the tool.

Lessons Learned

Analysis of incident

Following are the brief steps and flow of the reconnaissance that had taken place and the incident response steps that followed.

1. Attacker runs paratrace program with the target of the web server
2. Attacker then connects to web server, which the firewall passes as connections to the web server are allowed based on rule 1.
3. The paratrace program creates duplicates of the TCP packets on port eighty, the firewall passes them, as this connection is already in the state table of the firewall. Number of duplicates is equal to the number of hops to the target. Although the packets are not exact duplicates, the TTL value and IP ID values are modified to trigger responses.
4. Routing devices that see the TCP packets with TTL 1, decrement the value to 0, drop the packet and send an ICMP "Time Exceeded" message back to the originator of the TCP packets, the attacker.
5. Firewall passes these packets, as "Allow ICMP" is selected in the security policy of the firewall.
6. Attacker receives the ICMP messages and creates a map of the internal network up to the web server.
7. External IDS snort sensor alerted with "ICMP Time Exceeded" messages
8. Incident team went on site and obtained a forensic copy of the hard drive of the watcher server. Evidence and incident process was followed.
9. Incident team found the source of the attack and the stimulus packets from the attacker.
10. Incident team blocked the outgoing internal information from passing back to the attacker.
11. Reconnaissance tool was determined and located and tested against own network to determine if still vulnerable to further attacks.
12. Other methods of further strengthening security were investigated.

Recommendations

Cause of problem

Although a specific rule was put in the firewall to stop people mapping the network with ICMP echo requests, the paratrace technique was still successful. Investigation found that the main reason for this was the parameter in the security policy of the firewall which allowed ICMP messages through (Allow ICMP) even though there was no specific firewall rule permitting this. The containment control of adding a new rule in the firewall which specifically stopped ICMP "Time Exceeded" messages from leaving the internal network was effective to stop this reconnaissance technique.

Recommendation

The recommendation is to keep the rule of blocking outgoing ICMP "Time Exceeded" messages in the firewall. Secondly, it would be ideal to

deselect the “Allow ICMP” in the security policy of the firewall and investigate what ICMP traffic is required for proper operation of the network and specifically allow this through the firewall instead of allowing a carte blanche for all ICMP. The investigation should include current exploits, Trojans, DOS’s and backdoors available that utilise ICMP messages for operation.

Results of problem

The reconnaissance was successful in this case. The attacker gained information about the internal IP addresses of the network.

Recommendation

A review to be conducted to weigh up the risk of this knowledge being in the hands of an external person, or many depending on if the information has been shared and the cost of changing the internal network IP address so that the gained information is no longer valid. Secondly that the findings of the review be carried out.

Incident Process problem - 1

Possible problems could have arisen from blocking all outgoing ICMP “Time Exceeded” messages. No testing of the companies internet services were conducted from the point of view of an external person. Although there is probably a fairly low chance of this, it still would have been prudent to conduct some testing to help ensure no interruption to business services.

Recommendation

For any new change, the change should be tested in the test lab and all critical services tested before applying to the production network. Incident procedures to be changed to reflect this.

Incident Process problem - 2

Contrary to the policy, all systems did not have a synchronised clock. Therefore there were times during the incident where is was confusing which packet related to which other packet from different systems.

Recommendation 2

Synchronise all systems to one single time source. Audits to include checking this part of the policy is being carried out.

Incident Process problem - 3

The companies evidence handling process calls for a forensic imaging provider to be used to conduct the forensic imaging process. This was not

needed in this case and if used would have added hours to the whole process. As nothing should be done on the system until the forensic image has been completed this would have had the effect of leaving the path open for the reconnaissance tool to keep working for the attacker for a longer period, thereby allowing them to continue to map the network.

Recommendation 3

Change the evidence handling process to have a decision box in the part where a forensic image is required. The decision box would ask the question, is a mirrored disk available for use for the forensic image? If yes, then use that disk, otherwise call in the forensic image provider.

Incident Process well done - 1

Rapid response to the initial problem and confirmation that the alert was actually an incident that needed to be addressed.

Recommendation 1

Continue to have watcher server outside the firewall which captures all packets. If a secondary link to the internet is established, then a secondary packet capture server should also be implemented.

Incident Process well done - 2

Forensic imaging of device was done very easily and timely.

Recommendation 2

As the watcher server was deemed a critical part in determining possible causes of security incidents and because of this was initially marked as a likely source of evidence, the mirrored disk option was selected. Any time a forensic image was needed, it would be a simple matter to just pull one of the mirrored hard drives. A review should be instigated to determine other systems that may be used for evidence. Once these systems are selected, then they should be configured to have mirrored disks also if not already.

Costs

Security personal	1 day	
Onsite incident handler (*2)	1 day	
Administration questions	2 hours	
Review	2 days	
If recommendations followed		
Recommendation 1 –	2 days	
Recommendation 2 -	2 days	/ 4 days
Recommendation 3 –	1 day	
Recommendation 4 –	3 days	
Recommendation 5 –	2 hours	
Recommendation 6 –	N/A	
Recommendation 7 –	4 days	

Total time for incident	16.5 days	/ 20.5 days

Although the total time to address the incident was just over four man days, the total time to properly address the incident to the full extent, would be sixteen and a half days. Using the standard company rate of $100 an hour, the total cost of the incident is $13,300. If the results of the review of the internal network information being out in the public is to change the network then the total cost is $16,400.

Summary

The initial preparation for the incident was good and sufficient to deal with this case. The problem was addressed in a timely manner and the communication channel was sufficient. There were some problems overall and if the recommendations are followed then the next incident will be addressed more effectively and efficiently. This incident was a minor one of reconnaissance only and the total cost of the incident was $13,300 or $16,400 depending on if the internal network is fully re-secured.

Extras

A management report is to be produced. This is a one page report that contains a summary of the incidents that have happened for the month and the costs involved. This report should be distributed to the CIO and the board.

List of References

- Kaminsky, Dan, URL http://www.doxpara.com
- Kaminsky, Dan, “Paketto Brief” URL http://www.doxpara.com/read.php/docs/pk_english.html
- Kaminsky, Dan, “Paketto-1.1”. Version 1.1 URL http://www.doxpara.com/paketto/paketto-1.1.tar.gz
- Kaminsky, Dan, “BH-Asia-02-kaminsky.pdf” URL http://www.blackhat.com/presentations/bh-asia-02/Kaminsky/bh-asia-02-kaminsky.pdf
- DeJesus, Edmund X, “Paketto Keiretsu: New TCP/IP Tools Cut Both Ways “. URL http://www.infosecuritymag.com/2002/nov/digest25.shtml#news1
- Johnston, Gretel. “The Golden Age Rolls On”. URL http://www.computerworld.com/securitytopics/security/holes/story/0,10801,75381,00.html
- Baker, F. “RFC1812 Requirements for IP Version 4 Routers “ June 1995. URL http://www.ietf.org/rfc/rfc1812.txt?number=1812
- Postel, J. “rfc792 INTERNET CONTROL MESSAGE PROTOCOL“ September 1981. URL http://www.ietf.org/rfc/rfc0792.txt?number=792
- IETF, Braden, R. “rfc1122 Requirements for Internet Hosts -- Communication Layers” URL http://www.ietf.org/rfc/rfc1122.txt?number=1122
- Network Socery, “ICMP message 11” URL http://www.networksorcery.com/enp/protocol/icmp/msg11.htm
- Kaminsky, Dan. URL http://www.doxpara.com/read.php/code/paketto.html
- Kaminsky, Dan URL “Meet Dan Kaminsky IRC” http://umeet.uninet.edu/umeet2002/talk/2002-12-09/linux3.3.txt.html
- Kaminsky, Dan URL http://www.doxpara.com/read.php/docs/paratrace_logs.html
- USA, DOD, “DOD STANDARD - TRANSMISSION CONTROL PROTOCOL”
- Kaminsky, Dan. “Paketto” Version 1.1. URL http://www.doxpara.com/read.php/code/paketto.html

- URL http://www.google.com
- Kaminsky, Dan, “Black Ops Hivercon”. URL http://www.doxpara.com/Black_Ops_Hivercon.ppt
- “Snort Rule Stabe” URL http://www.snort.org/dl/rules/snortrules-stable.tar.gz

Definitions

FTP	- File Transfer Program
HTML	- Hyper Text Markup Language
ICMP	- Internet Control Message Protocol
IDS	- Intrusion Detection System
IP	- Internet Protocol
IP ID	- ID field in the IP header
NIDS	- Network Intrusion Detection System
TTL	- Time To Live
TCP	- Transmission Control Protocol
Tcpdump	- Program to capture packets
UDP	- User Data Protocol

Appendix 1

PARATRACE(1) PARATRACE(1)

NAME

paratrace - Parasitic Traceroute via Established TCP Flows & IPID Hopcount

SYNOPSIS

paratrace [-b bandwidth] [options] host|network

PACKAGE

Paketto Keiretsu 1.0

DESCRIPTION

Paratrace traces the path between a client and a server, much like "traceroute", but with a major twist: Rather than iterate the TTLs of UDP, ICMP, or even TCP SYN packets, paratrace attaches itself to an existing, stateful-firewall-approved TCP flow, statelessly releasing as many TCP Keepalive messages as the software estimates the remote host is hop-distant. The resultant ICMP Time Exceeded replies are analyzed, with their original hop-count "tattooed" in the IPID field copiedinto the returned packets by so many helpful routers. Through this process, paratrace can trace a route without modulating a single byte of TCP/Layer 4, and thus delivers fully valid (if occasionally redundant) segments at Layer 4 -- segments generated by another process entirely.

OPTIONS

-s [hopfuzz]

"Overshoot" target host by a number of hops equal to the hopfuzz. paratrace passively estimates the number of hops away that a given target is, based on the incoming ACK/PSH|ACK's TTL's deviation from a clean factor of 64. This is, at *best*, a rough estimate, so we set the number of hops larger to make sure we get all the way there. (The scan is stateless, so we can't just send more packets when we don't get all we wanted.) Setting this value to -1 may offer some degree of stealth.

-t [number of seconds]

Set maximum number of seconds that may pass before listening process gives up on receiving any more responses. This timer is reset with every good response, whether the port is up or down.

-b [bandwidth][b][k][m][g]
Limit the amount of bandwidth that scanrand may use for its outgoing requests. -b 100k would limit said bandwidth to 100kbyte/s. Note, since outgoing SYN frames constitute only 64 bytes on the wire, very little bandwidth can go very, very far. The bandwidth value of 0 -- set by default -- corresponds to no bandwidth limitation.

-n Specify network, instead of host, to attach a parasitic trace to. This has the caveat ofrequiring an IP address, rather than a single DNS name, but in return CIDR notation(1.2.3.0/24) lets users specify pretty decent host ranges they're interested in.

-N Use Reverse-DNS to determine a DNS host name that matches the source of a detected packet.

-NN Use Reverse-DNS to determine a DNS host name that matches the intended destination of a given packet.

-v Verbosity Level 1: Mark the sending of packets.

-vv Verbosity Level 2: Output all interpreted TCP and IP headers.

ADDRESSING
-d <interface>
Use this Layer 2 Device for all traffic.

-i <source ip>
Use this Layer 3 Source IP for all traffic.

BUGS
Again, it is very easy to scan faster than the network can accept packets.No code exists to dynamically reduce scan speed. Furthermore, we can only capture absolute latency from beginning of scan, rather than latency of a given response to its received packets. There is a fix in progress for this. Also, bandwidth calculation presently doesn't take into account the time necessary to actually send a given packet. This will be fixed.

AUTHOR

This work has been done by Dan Kaminsky of DoxPara Research, who may be reached at dan@doxpara.com.

PARATRACE(1)

Appendix 2 – paratrace.c

```
#include "paketto.h"
#include "pk_crypt.h"
#include "scanutil.h"
/*#include "d_services.h"*/

void paratrace_usage();

int main(int argc, char **argv)
{
  int opt;
  extern char *optarg;
  extern int   opterr;

  pcap_t *pcap;               /* PCAP descriptor */
  u_char *packet;             /* Our newly captured packet */
  char pfprogram[2048];
  char dev[255];
  char *p;
  char target[1024];
  long source_ip  = 0;
  int source_port = 0;
  struct pcap_pkthdr pkthdr;        /* PCAP packet information structure */
  struct bpf_program fp;    /* Structure to hold the compiled prog */
  char error[PCAP_ERRBUF_SIZE];               /* Structure for libpcap errors.
*/

  struct frame x, ic;

  int hopfuzz = 4;
  int network = 0;
  int up = 0;
  int immediate = 1;
  int i,j,k,l,pid;
  float timeout = 60;
  int verify=1;
  int force_sip  = 0;
  int resolve = 0;

  int verbose = 0;

  long li,lj;
  struct in_addr temp_ip;

  char buf[MX_B], buf2[MX_B], destbuf[1024], rangebuf[1024], portbuf[1024];
  char dest[MX_B];
  char *ttlrange = NULL;
  char *bandwidth = NULL;
```

```
  int check_icmp_seq = 0;
  int target_acquired = 0;
  u_char *seed = malloc(20);

  u_char *tcpscan = malloc(MX_B);
  struct libnet_link_int *temp = NULL;

  struct frame *scanx;

  struct timeval start, now, then, diff;

  FILE *targets, *logs;

  prng_state prng;
  pk_initrng(&prng);

  bzero(buf, sizeof(buf));
  bzero(buf2, sizeof(buf2));

      if(geteuid() != 0)
      {
            perror("PK requires root to access the network directly.");
            exit(EXIT_FAILURE);
      }

  p = NULL;
  p = pcap_lookupdev(error);
  if(!p){
    fprintf(stderr, "Couldn't lookup default ethernet device with
pcap_lookupdev: %s\n", error);
    exit(EXIT_FAILURE);
  }
  snprintf(dev, sizeof(dev), "%s", pcap_lookupdev(error));

  while ((opt = getopt(argc, argv, "d:i:nNt:b:T:vs:S")) != EOF) {
    switch (opt) {
      case 'd':
        snprintf(dev, sizeof(dev), "%s", optarg);
        break;
      case 'i':
        source_ip = ntohl(libnet_name_resolve(optarg, 0));
        force_sip++;
        break;
      case 'n':
        network++;
        break;
        case 'N':
          resolve++;
          break;
        case 't':
```

```
            timeout = atof(optarg);
            break;
        case 'b':
          bandwidth = malloc(1024);
          snprintf(bandwidth, 1024, "%s", optarg);
          break;
          case 'v':
            verbose++;
            break;
          case 's':
            hopfuzz = atoi(optarg);
            break;
          case 'S':
            hopfuzz = -1;
            break;
          default:
            paratrace_usage();
       }
  }

  if(argv[optind] != NULL)
  {
        snprintf(target, sizeof(target), "%s", argv[optind]);
  } else {
        fprintf(stderr, "Paratrace requires a target to attempt a trace
against.\n");
        paratrace_usage();
  }

  if(!force_sip) source_ip=libnet_get_ipaddr(temp, dev, NULL);

  if(!bandwidth){
        bandwidth = malloc(1024);
        snprintf(bandwidth, 1024, "0");
        }


  if(verbose)fprintf(stderr,
"Stat|=====IP_Address==|Port=|Hops|==Time==|============Details===
=========|\n");
          fprintf(stderr, "Waiting to detect attachable TCP connection to
host/net: %s\n", target);
  gettimeofday(&start, NULL);

  if(!network) snprintf(buf, sizeof(buf), "host");
  else         snprintf(buf, sizeof(buf), "net");
  snprintf(pfprogram, sizeof(pfprogram), "icmp or (src %s %s and tcp)", buf,
target);
  pcap = pcap_open_live(dev, 65535, 1, 1, error);
  if(!pcap){
```

```
          fprintf(stderr, "Couldn't open device: %s\n", error);
          exit(1);
         }

  ioctl(pcap_fileno(pcap), BIOCIMMEDIATE, &immediate);  // prolly breaks
nonblock

  if (pcap_compile(pcap, &fp, pfprogram, 1, 0x0) == -1) {
    pcap_perror(pcap, "pcap_compile");
    exit(EXIT_FAILURE);
  }

  if (pcap_setfilter(pcap, &fp) == -1) {
    pcap_perror(pcap, "pcap_setfilter");
    exit(EXIT_FAILURE);
  }
  gettimeofday(&now, NULL);
  gettimeofday(&then, NULL);

  while(!timeout || now.tv_sec <= (then.tv_sec + timeout))
  {
    packet = (u_char *) pcap_next(pcap, &pkthdr);
    gettimeofday(&now, NULL); /* packet header sigfigs seem strange */
    if(packet &&
       parse_layers(packet, pkthdr.caplen, &x, 2, pcap_datalink(pcap), 0)){

          /* Accept ICMP packets. */

          if(target_acquired &&
             x.ip->ip_p == IPPROTO_ICMP){
             i=parse_layers((char *)&x.icmp->icmp_data,
             pkthdr.caplen-LIBNET_ETH_H-(int)x.ip->ip_hl*4-LIBNET_ICMP_H,
/* XXX slight chance of bug */
             &ic, 3, DLT_EN10MB, 1);
          if(i && ic.ip->ip_p == IPPROTO_TCP &&
             x.icmp->icmp_type == ICMP_TIMXCEED){
                    timeval_subtract(&diff, &pkthdr.ts, &start);
                    gettimeofday(&then, NULL); /* just for the loop
maintenance */
                if(verbose){
                     fprintf(stderr, "Got %i on %s:\n", pkthdr.caplen, dev);
                     fprintf(stderr, " "); print_ip((char *)x.ip);
                     fprintf(stderr, "ICMP: "); print_ip((char *)ic.ip);
                     fprintf(stderr, "ICMP: "); print_tcp((char *)ic.tcp, 1);
                     }
                  bzero(buf, sizeof(buf));
                    bzero(buf2, sizeof(buf2));
                  /* whoa this works?! WTF */
                    snprintf(buf + 0, 16, inet_ntoa(x.ip->ip_src));
                    snprintf(buf +16, 16, inet_ntoa(ic.ip->ip_src));
```

```
                        snprintf(buf +32, 16, inet_ntoa(ic.ip->ip_dst));
                        //fprintf(stdout, "%3.3u = ", 255 - (source_port -
ntohs(ic.tcp->th_sport)));
                          fprintf(stdout, "%3.3u = ", htons(ic.ip->ip_id));
                          fprintf(stdout, "%16.16s|%-5i [%2.2hu]", buf,
                                ntohs(ic.tcp->th_dport), estimate_hopcount(x.ip-
>ip_ttl));
                          fprintf(stdout, "%4lu.%3.3lus", diff.tv_sec,
diff.tv_usec/1000);
                          if(resolve==1)
                            fprintf(stdout, "(%35.35s)\n", libnet_host_lookup(x.ip-
>ip_src.s_addr, 1));
                          else if(resolve==2)
                            fprintf(stdout, "(%35.35s)\n", libnet_host_lookup(ic.ip-
>ip_dst.s_addr, 1));
                          else fprintf(stdout, "(%16.16s -> %-16.16s)\n", buf+16,
buf+32);
                  }
              else if(target_acquired &&
                    x.icmp->icmp_type == ICMP_UNREACH &&
                    ic.ip->ip_p == IPPROTO_TCP)
                    {
                        timeval_subtract(&diff, &pkthdr.ts, &start);
                        gettimeofday(&then, NULL); /* just for the loop
maintenance */
                        snprintf(buf2, sizeof(buf2), "un%2.2i", x.icmp-
>icmp_code);
                        snprintf(buf + 0, 16, inet_ntoa(x.ip->ip_src));
                        snprintf(buf +16, 16, inet_ntoa(ic.ip->ip_src));
                        snprintf(buf +32, 16, inet_ntoa(ic.ip->ip_dst));
                        fprintf(stdout, "%s: %16.16s:%-5i [%2.2hu]",
                            buf2, buf+32, ntohs(ic.tcp->th_dport),
                            estimate_hopcount(x.ip->ip_ttl));
                        fprintf(stdout, "%4lu.%3.3lus", diff.tv_sec,
diff.tv_usec/1000);
                        if(resolve==1)
                          fprintf(stdout, "(%35.35s)\n", libnet_host_lookup(x.ip-
>ip_src.s_addr, 1));
                        else if(resolve==2)
                          fprintf(stdout, "(%35.35s)\n", libnet_host_lookup(ic.ip-
>ip_dst.s_addr, 1));
                        else      fprintf(stdout, "(%16.16s -> %-16.16s)\n",
buf+16, buf);
                    }

          }
          /* Accept...ummm...anything TCP from target. */
            else if(!up && target_acquired &&
                  x.ip->ip_p == IPPROTO_TCP)
                  {
```

```
            if(verbose>=2){
                  fprintf(stderr, "Got %i on %s:\n", pkthdr.caplen, dev);
                  fprintf(stderr, " "); print_ip((char *)x.ip);
                  fprintf(stderr, " "); print_tcp((char *)x.tcp, 0);
                  }
                gettimeofday(&then, NULL); /* just for the loop maintenance */
                bzero(buf, sizeof(buf));
                bzero(buf2, sizeof(buf2));
                timeval_subtract(&diff, &now, &start);
                snprintf(buf2, sizeof(buf2), "  UP"); /* we got SOMETHING */
                if((int)buf2[0]) /* :-P */
                  {
                  fprintf(stdout, "%s: %16.16s:%-5i [%2.2hu]", buf2,
inet_ntoa(x.ip->ip_src), ntohs(x.tcp->th_sport), estimate_hopcount(x.ip-
>ip_ttl));
                  fprintf(stdout, "%4lu.%3.3lus", diff.tv_sec, diff.tv_usec/1000);
                  if(resolve)fprintf(stdout, "(%35.35s)\n", libnet_host_lookup(x.ip-
>ip_src.s_addr, 1));
                  else      fprintf(stdout, "\n"); /*fprintf(stdout, "(%29.29s)\n", buf);
*/
                  }
                up++;
                //exit(0); /* gotta figure out to precisely detect the response */
          }
        /* Got an ACK?  Lets trace it back w/ a keepalive, which'll pass all dem
stateless filters */
          /* I haven't figured out yet how to detect *hitting* the actual target vs.
normal traffic */
          else if(!target_acquired &&
                  x.ip->ip_p == IPPROTO_TCP &&
                  x.tcp->th_flags == TH_ACK)
                {
                char temp_mac[ETHER_ADDR_LEN];

                target_acquired++;
                if(!ttlrange) ttlrange = malloc(1024);
                snprintf(ttlrange, 1024, "1-%u", estimate_hopcount(x.ip-
>ip_ttl)+hopfuzz);

                snprintf(dest, sizeof(dest), "%s:%u/32", inet_ntoa(x.ip->ip_src),
ntohs(x.tcp->th_sport));

                pk_memswp(&(x.eth->ether_dhost), &(x.eth->ether_shost),
ETHER_ADDR_LEN);
                pk_memswp(&(x.ip->ip_src), &(x.ip->ip_dst),
IPV4_ADDR_LEN);
                pk_memswp(&(x.tcp->th_sport), &(x.tcp->th_dport), 2);
                pk_memswp(&(x.tcp->th_seq), &(x.tcp->th_ack), 4);

                /* zero the payload */
```

```
                x.ip->ip_len = htons((int)x.ip->ip_hl*4 + (int)x.tcp->th_off*4);
                pkthdr.caplen = LIBNET_ETH_H + (int)x.ip->ip_hl*4 + (int)x.tcp-
>th_off*4;
                recalc_checksums(&x, IPPROTO_TCP);

                fprintf(stderr, "%s %s\n", dest, ttlrange);
                pid=0;
                pid=fork();
                if(!pid){
                        usleep(1000); /* wait for OS to deal with that segment first
*/
                        raw_sock_syn_scan(dest, sizeof(dest), dev, &x,
                                        ttlrange, seed, bandwidth, verbose, resolve,
1);
                        exit(0);
                        }
                }
        }

    }
  }
void paratrace_usage()
{
  fprintf(stderr, "Paratrace %s: Parasitic Traceroute via Established TCP
Flows & IPID Hopcount\n", VERSION);
  fprintf(stderr, "Component of:  Paketto Keiretsu %s;    Dan Kaminsky
(dan@doxpara.com)\n\n", VERSION);
  fprintf(stderr, "     Example:  Paratrace -b100k www.doxpara.com\n");
  fprintf(stderr, "     Example:  Paratrace -t0 -n 10.0.1.0/24\n");
  fprintf(stderr, "     Options:  -s  [hopfuzz]: Fuzz hopcount estimation for TTL
scan       (+4)\n");
  fprintf(stderr, "             -t  [timeout]: Wait n full seconds for the last response
(60s)\n");
  fprintf(stderr, "             -b[bandwidth]: Limit bandwidth consumption to n
b/k/m/g bytes(0)\n");
  fprintf(stderr, "                          (0 supresses timeouts; maximizes bw
utilization)\n");
  fprintf(stderr, "             -n          : Specify network instead of host to
respond to\n");
  fprintf(stderr, "             -N/-NN      : Enable name resolution (Prefer
Source/Dest)\n");
  fprintf(stderr, "             -v          : Mark packets being sent, as well as
received\n");
  fprintf(stderr, "             -vv         : Output full packet traces to stderr\n");
  fprintf(stderr, "  Addressing:  -d   [device]: Send requests from this L2
hardware device\n");
  fprintf(stderr, "             -i   [source]: Send requests from this L3 IP
address\n");
  exit(1);
}
```

Appendix 3 – scanutil.c

```
#include <scanutil.h>
#include <d_services.h>

long bake_syncookie(u_char *ipp, u_char *key)
{
  u_char buf[MX_B];
  struct libnet_ip_hdr *ip = NULL;
  struct libnet_tcp_hdr *tcp = NULL;

  u_char syncookie[20];
  long synbits;

  (char *)ip  = (char *)ipp;
  (char *)tcp = (char *)ip + (int)ip->ip_hl*4;


   bzero(buf, sizeof(buf));
   memcpy(buf,   &ip->ip_src, 4);
   memcpy(buf+4, &ip->ip_dst, 4);
   memcpy(buf+8, &tcp->th_sport, 2);
   memcpy(buf+10,&tcp->th_dport, 2);

  pk_hmac(syncookie, key, buf, 12);
  memcpy(&synbits, &syncookie, sizeof(synbits));

  /*fprintf(stderr, "Sending: %lx vs. %lx from %i/%i\n", ntohl(synbits), 0,
    ntohs(tcp->th_sport), ntohs(tcp->th_dport));*/

  return(synbits);
}

long munch_syncookie(u_char *ipp, u_char *key)
{
  u_char buf[MX_B];
  struct libnet_ip_hdr *ip = NULL;
  struct libnet_tcp_hdr *tcp = NULL;
  int i = 1;

  u_char syncookie[20];
  long synbits;

  (char *)ip  = (char *)ipp;
  (char *)tcp = (char *)ip + (int)ip->ip_hl*4;

  bzero(buf, sizeof(buf));
  memcpy(buf,   &ip->ip_dst, 4);
```

```
  memcpy(buf+4, &ip->ip_src, 4);
  memcpy(buf+8, &tcp->th_dport, 2);
  memcpy(buf+10,&tcp->th_sport, 2);

  pk_hmac(syncookie, key, buf, 12);

  memcpy(&synbits, &syncookie, sizeof(synbits));

  if(tcp->th_flags == TH_RST) /* Distco st00 rides again */
  {
    if(!(ntohl(tcp->th_ack))) tcp->th_ack = tcp->th_seq; /* WEIRD fix */
    i=0;/* RST's don't increment */
  }

  if(ntohl(tcp->th_ack)-i == ntohl(synbits))
  {
        return(synbits);
  }

  else{
        return(0); // this screws up 1/2^32 times.
  }
}

/* XXX shaddup shaddup i know raw_sock_syn_scan is horrifying, shaddup */

int raw_sock_syn_scan(char *dest, int length, char *dev, struct frame *scanx,
                char *ttlrange, char *seed, char *bandwidth, int verbose, int
resolve,
                int disable_seq)
{
  char abuf[1024], bbuf[1024], cbuf[1024], dbuf[1024], pbuf[1024], tbuf[1024],
rbuf[1024];
  unsigned short a, b, c, d, ttl, start_a, end_a, start_b, end_b;
  unsigned short start_c, end_c, start_d, end_d, start_p, end_p;
  unsigned short start_ttl, end_ttl;
  unsigned int dport;

  unsigned int base, multiple, packetsleep;
  int knownscan = 0;
  int kcount = 0;
  int flag = 0;

  struct timeval now, then, bench_pre, bench_post, diff;

  struct libnet_link_int *temp = NULL;
  int sockfd = -1;
  int i,j,source_port = ntohs(scanx->tcp->th_sport);
  int keep_ipid = 0;
  char buf[MX_B], buf2[MX_B];
```

```
  struct in_addr temp_ip;
  struct libnet_plist_chain *alist, *blist, *clist, *dlist, *plist, *tlist;

  sockfd = libnet_open_raw_sock(IPPROTO_RAW);

  gettimeofday(&then, NULL);

  if(!sockfd)
  {
        fprintf(stderr, "Couldn't open raw socket.\n");
        exit(1);
  }

  /* We need to figure out how fast we're allowed to send packets.  We do this
   * by noting that our standard packet is 40 bytes, plus 14 from ethernet,
yielding
   * 54 bytes on the wire.  Ah, but the minimum size for an ethernet frame is
64 bytes,
   * and until I get off my duff and properly support non-ether interfaces, that'll
   * have to be our per-packet cost.
   */


  if(!bandwidth || bandwidth[0]=='0') packetsleep=0;
  else{
        i=sscanf(bandwidth, "%1024[^BbKkMmGg]%1024s", buf, buf2);
        if(i==0)return(0);
        base=atoi(buf);
        if(i==1){buf2[0]='B'; i=2;}
        if(i==2)switch(buf2[0]){
                case 'B':
                        multiple=1;
                        break;
                case 'b':
                        multiple=1;
                        break;
                case 'K':
                        multiple=1024;
                        break;
                case 'k':
                        multiple=1024;
                        break;
                case 'M':
                        multiple=1024*1024;
                        break;
                case 'm':
                        multiple=1024*1024;
                        break;
                case 'G':
```

```
                        multiple=1024*1024*1024;
                        break;
                case 'g':
                        multiple=1024*1024*1024;
                        break;
                }
        i=base*multiple;
    /* XXX Need to incorporate time spent actually sending packets :-) */
        packetsleep=(1000000*64)/i; /* 64 is minimum frame size */
  }
  if(sscanf(dest,
"%1024[^.].%1024[^.].%1024[^.].%1024[^:]:%1024[^/]%1024s",
          abuf, bbuf, cbuf, dbuf, pbuf, rbuf) != 6) return(0);

  if(!ttlrange){
        ttlrange = malloc(1024);
        snprintf(ttlrange, 1024, "%i-%i", scanx->ip->ip_ttl, scanx->ip->ip_ttl);
 }
  if(!strncmp(pbuf, "known", sizeof(pbuf))){
          knownscan++;
          snprintf(pbuf, sizeof(pbuf), "0-1");
  }

  libnet_plist_chain_new(&alist, abuf);
  while(libnet_plist_chain_next_pair(alist, &start_a, &end_a)){
   libnet_plist_chain_new(&blist, bbuf);
   while(libnet_plist_chain_next_pair(blist, &start_b, &end_b)){
    libnet_plist_chain_new(&clist, cbuf);
    while(libnet_plist_chain_next_pair(clist, &start_c, &end_c)){
     libnet_plist_chain_new(&dlist, dbuf);
     while(libnet_plist_chain_next_pair(dlist, &start_d, &end_d)){
      libnet_plist_chain_new(&plist, pbuf);
      while(libnet_plist_chain_next_pair(plist, &start_p, &end_p)){
       libnet_plist_chain_new(&tlist, ttlrange);
       while(libnet_plist_chain_next_pair(tlist, &start_ttl, &end_ttl)){
        /* libnet_plist was meant for port lists, but we're hacking it
           to do IP/TTL lists as well.  Though an IP is 32 bytes, ports are
           16 bytes, and each range is an 8 byte range.  So, we clamp the
           iteration to an 8 byte range. */
        if(start_a > 255) start_a = 255;     if(end_a > 255) end_a = 255;
        if(start_b > 255) start_b = 255;     if(end_b > 255) end_b = 255;
        if(start_c > 255) start_c = 255;     if(end_c > 255) end_c = 255;
        if(start_d > 255) start_d = 255;     if(end_d > 255) end_d = 255;
        if(start_ttl > 255) start_ttl = 255; if(end_ttl > 255) end_ttl = 255;

        //fprintf(stderr, "%u-%u.%u-%u.%u-%u.%u-%u:%u-%u\n", start_a,
end_a, start_b,
          //        end_b, start_c, end_c, start_d, end_d, start_p, end_p);
```

```
        for(ttl=start_ttl; ttl<=end_ttl; ttl++){
        for(a=start_a; a<=end_a; a++){
        for(b=start_b; b<=end_b; b++){
        for(c=start_c; c<=end_c; c++){
          //usleep(subnet_sleep*1000);
          gettimeofday(&now, NULL);
        for(d=start_d; d<=end_d && d; d++){
        for(dport=start_p; dport<=end_p; dport++){

            scanx->ip->ip_dst.s_addr = ntohl(a*256*256*256 + b*256*256 +
c*256 + d);
          /*bzero(buf, sizeof(buf));
          snprintf(buf, sizeof(buf), "%u.%u.%u.%u", a, b, c, d);
          inet_aton(buf, &scanx->ip->ip_dst);*/ /* cheap trick */
            if(!disable_seq){
                  scanx->tcp->th_dport = htons(dport);
            scanx->tcp->th_sport = htons(source_port - 255 + ttl ); /* XXX i
know, i know -- this needs to be time */
            }
          if(knownscan){
                    scanx->tcp->th_dport = htons(knownports[kcount].port);
                    if(kcount<1150)dport=0;
                    else          dport=1;
                    kcount++;
            }
            scanx->ip->ip_ttl = ttl;
          scanx->ip->ip_id =  htons(ttl); /* redundant hop capacity, for your
convenience */

          if(!disable_seq){
                  i=bake_syncookie((u_char *)scanx->ip, seed);
                  memcpy(&scanx->tcp->th_seq, &i, 4);
                    if(scanx->tcp->th_flags == TH_ACK){
                    memcpy(&scanx->tcp->th_ack, &i, 4);
                    }
            }

          recalc_checksums(scanx, scanx->ip->ip_p);
          i = libnet_write_ip(sockfd, (char *)scanx->ip, (int)scanx->ip->ip_hl*4 +
(int)scanx->tcp->th_off*4);
          if(verbose>=1){
                  gettimeofday(&now, NULL);
                  timeval_subtract(&diff, &now, &then);
                  fprintf(stdout, "%s: %16.16s:%-5i [%2.2hu]", "SENT",
inet_ntoa(scanx->ip->ip_dst), ntohs(scanx->tcp->th_dport), 0);
                  fprintf(stdout, "%4lu.%3.3lus", diff.tv_sec, diff.tv_usec/1000);
                  if(resolve)fprintf(stdout, "(%35.35s)\n",
libnet_host_lookup(scanx->ip->ip_dst.s_addr, 1));
                  else      fprintf(stdout, "\n"); /*fprintf(stdout, "(%29.29s)\n", buf);
*/
```

```
                }
            if(verbose>=2){
                fprintf(stderr, "Sent %i on %s:\n", i, dev);
                fprintf(stderr, " "); print_ip((char *)scanx->ip);
                fprintf(stderr, " "); print_tcp((char *)scanx->tcp, 0);
                }
              if(packetsleep)usleep(packetsleep);
          }}}}}} /* all those for loops */
         }libnet_plist_chain_free(tlist);
        } libnet_plist_chain_free(plist);
       }  libnet_plist_chain_free(dlist);
      }   libnet_plist_chain_free(clist);
     }    libnet_plist_chain_free(blist);
    }     libnet_plist_chain_free(alist);
    return(0);
}

struct frame *build_generic_syn(struct frame *x)
{
  x->data = malloc(2048);

  x->eth = x->data;

  libnet_build_ethernet("000000",     /*x.eth->ether_dhost*/
                "000000",  /*x.eth->ether_shost*/
                ETHERTYPE_IP, /*x.eth->ether_type*/
                NULL,         /*extra crap to tack on*/
                0,            /*how much crap*/
                (char *)x->eth);

  (char *)x->ip = (char *)x->eth + LIBNET_ETH_H;

  libnet_build_ip(LIBNET_TCP_H,
            0, // tos
            1234, // ipid
            0,    // frag
            255,  // ttl
            IPPROTO_TCP,
            0, //source
            0, // dest
            NULL, // ip payload
            0,    // ip payload size
            (char *)x->ip);

  x->ip->ip_off = 64; /* set DF flag */
  (char *)x->tcp = (char *)x->ip + (int)x->ip->ip_hl*4;

  libnet_build_tcp(12345, // source port
             139,  // dest port
             420, // seq
```

```
                0, // ack
                TH_SYN, // flags
                4096, // win
                0, // urgp
                NULL, // tcp payload
                0, // tcp payload size
                (char *)x->tcp);
  return(x);
}

int estimate_hopcount(int ttl)
{
  /* tip of the hat to nomad's despoof -- no, this ain't supposed to be perfect */
  int passive_factor=32; /* may be low but i found a host w/ base TTL 32 */
                  /* i've heard rumors of hosts w/ ttl base 240 but
                     haven't found any yet */
                  /* when wrong, it'll usually be off by 4 (damn 60 ttl hosts)*/
  int distco_factor = 216; /* any less and we hit win32 ttl=128, any more and
valid */
  if(ttl < (distco_factor-80) || ttl > distco_factor) /* handles up to 40 hops */
  {
    if(ttl%passive_factor == 0)ttl--;
    return(passive_factor - (ttl%passive_factor));
  } else {
    return((distco_factor - ttl + 1) / 2); /* they don't adjust ttl when they RST! */
  }
}

int parse_dest(char *dest, int length, char *shortdest, int multi)
{
     char buf[MX_B], buf2[MX_B], destbuf[1024], rangebuf[1024],
portbuf[1024];
     int i,j,k,l;
     struct in_addr temp_ip;
     const char quickbuf[] = "80,443,445,53,20-
23,25,135,139,8080,110,111,143,1025,5000,465,993,31337,79,8010,8000,66
67,2049,3306";
     const char squickbuf[] = "80,443,139,21,22,23";


     /* there's gotta be an easier way to do this :-)
      * basically we're splitting the destination string into
      * destination, CIDR range, and ports...the whole purpose of
      * this function is to take whatever garbage the user gave us
      * and convert it to my canonical form -- or die trying.
      */
     sscanf(shortdest, "%1024[^/]/%1024s", buf, rangebuf); /* only need range
*/
     sscanf(shortdest, "%1024[^:/]:%1024[^:/]:%1024s", destbuf, portbuf, buf);
```

```
	if(!destbuf[0]){
	return(0);
	}

    /* So, here's the deal.  By default, I want to scan the bejesus out of a single
host, but
	 I don't want to throw out thousands of packets per host for a simple
	 traceroute or netsweep.  So there's differential default behavior, but it's
doing
	 the Right Thing.  Manual override of course can do anything. */

	i=0;
	k=0;

	while(i<1024 && destbuf[i]!=0){
		if((destbuf[i]=='-' || destbuf[i]==',')){
			k=2; /* might be plural */
		}
		else if((destbuf[i]>='A' && destbuf[i]<='Z') ||
			 (destbuf[i]>='a' && destbuf[i]<='z')){
			j++;
		}
		if(destbuf[i]=='\n' || destbuf[i] == '\r'){
		    destbuf[i]=0; /* chomp, DNS resolver needs no trailing newline */
			}
		i++;
	}

	   if(k==2 && !j) multi=2;/* found dash/comma w/o DNS -- must be multi */
	   if(j){ /* found ASCII character */
		temp_ip.s_addr = libnet_name_resolve(destbuf, 1);
		  if(!memcmp(&temp_ip.s_addr, "\xff\xff\xff\xff",
IPV4_ADDR_LEN)){
			//fprintf(stderr, "Couldn't resolve name: %s\n", destbuf);
			return(0);
		  } else  snprintf(destbuf, 1024, "%s", inet_ntoa(temp_ip));
	   }

	if(sscanf(destbuf, "%1024[^.].%1024[^.].%1024[^.].%1024s",
buf,buf,buf,buf) != 4){
	   fprintf(stderr, "Invalid IP Specification:  Not enough octets(four
needed).\n");
	    return(0);
	}

	if(!atoi(rangebuf) || atoi(rangebuf)>32 || j+k==2) snprintf(rangebuf,
sizeof(rangebuf), "32");
	if(atoi(rangebuf) < 32) multi=3;

	if(!portbuf[0]){  /* no default ports?  Whatever shall we do! */
```

```
            if(multi) snprintf(portbuf, sizeof(portbuf), "80");
            else      snprintf(portbuf, sizeof(portbuf), "quick"); /* was d */
        }
        /* ok lets set up some defaults */
        if(!strncmp(portbuf, "quick", sizeof(portbuf))){
          snprintf(portbuf, sizeof(portbuf), "%s", quickbuf);
        }
        if(!strncmp(portbuf, "squick", sizeof(portbuf))){
             snprintf(portbuf, sizeof(portbuf), "%s", squickbuf);
        }
        if(!strncmp(portbuf, "all", sizeof(portbuf))){
             snprintf(portbuf, sizeof(portbuf), "0-65535");
        }
        /* we set up knownscan in raw_sock itself */

        snprintf(dest, length, "%s:%s/%s", destbuf, portbuf, rangebuf);
        return(1);
}
```