CITI Technical Report 98–6

Sifting the Network: Performing Packet Triage with NFR

M. Undy
mundy@umich.edu

C.J. Antonelli
cja@umich.edu

ABSTRACT

We describe a set of Network Flight Recorder scripts designed to detect network intrusions. After developing scripts that detect some known attacks, we focus on *sifting* scripts that attempt to remove "normal" traffic from a packet stream, leaving behind fewer packets requiring manual inspection. We conclude with a description of our experiences with the NFR product.

November 23, 1998

Sifting the Network: Performing Packet Triage with NFR

M. Undy
`mundy@umich.edu`

C.J. Antonelli
`cja@umich.edu`

INTRODUCTION

Passive protocol analysis is useful in detecting attacks on networked machines. Known attacks can be detected directly by looking for them in the network packet stream, using a network monitoring *protocol analysis engine* to look for unique bit patterns or sequences of packets unique to such attacks. Unknown attacks can be detected indirectly by separating for manual analysis suspicious traffic from traffic generated by customary network applications. For example, artificially fragmented packets, out-of-order TCP segments, overlapping segments, and packets containing incorrectly presented options generally do not occur during "normal" operations. A protocol analysis engine that sifts such "suspicious" packets from the network for manual inspection, eliding the normal traffic, is successful if the majority of network traffic is normal.

Filters for known attacks are useful in today's environment of widely-distributed attack scripts for rapidly identifying the existence of an attack and narrowing the space an administrator must examine before coping with the attack. For example, even if source addresses are forged, knowing that a "Ping of Death" attack is occurring allows an administrator to filter all ICMP echo request packets at the firewall for some period of time, or provide cumulative evidence that pings should be filtered permanently.

In this work we describe a protocol analysis engine both to detect known attacks and to sift the packet stream for suspicious traffic. We have selected the Network Flight Recorder [1] as a suitable development base, since it possesses an expressive scripting language, runs on UNIX platforms, and is available in source code form.

Ptacek and Newsham point out flaws in passive protocol analysis that can force a network monitoring machine to see packets that other hosts on the same network do not, prevent a network monitor from seeing traffic directed at other hosts, or attack a network monitor directly [2]. Nevertheless, we believe a protocol analysis engine is useful, within the limitations they outline, as their attacks are not as well-known as and are harder to construct than the attacks being monitored, particularly when faced with multiple analysis engines running on different operating system platforms.

OVERVIEW OF NFR

NFR consists of a number of components, each responsible for a specific activity:

- *Packet suckers* capture packets from the network interface.
- A *decision engine,* written in the N-code scripting language, checks packets against a list of filters.
- *Backends* forward selected packets to storage or statistical processing.
- A *query interface* permits examination of stored data.

NFR is a multi-purpose network monitoring tool, usable for intrusion detection, usage analysis, and troubleshooting by system administrators and the hacker community alike. The interpreted N-code language allows a user to write arbitrarily complex scripts for analyzing incoming packets, limited only by the timing constraints imposed by the NFR engine to ensure that filters share system resources fairly. To ease the task of the script writer, the N-code language includes primitives for analyzing packet header and data fields, and the NFR engine supports internal operations such as TCP reassembly. Functions called within the scripts pipe data to user specified backend databases. The graphical query interface uses HTML and Java to query these databases. Summarizing the NFR data flow, network packets are captured, N-code scripts are applied, and script outputs are sent to backends for storage and other processing.

NFR Testbed

We constructed an intrusion detection testbed, consisting of a 90 Mhz Pentium with 32 MB RAM, 1.2 GB disk, and a 3Com 3C509 Ethernet card connected to the CITI production network via a switch port configured to monitor all network packets.

After some initial problems with the NFR development version running on OpenBSD 2.2, we settled on running the 1.6.2 release of NFR on FreeBSD 2.2.6-RELEASE. Along the way we identified an error in the development version, which we passed along to NFR, Inc.

KNOWN-ATTACK SCRIPTS

After coordinating with NFR, Inc. staff to avoid duplication of effort, we wrote five known-attack scripts. The scripts themselves contain more detailed commentary. See Appendix A.

Smurf attack

This filter originated as a vehicle for learning how to write N-code. The filter detects the classic Smurf attack, looking for ICMP echo request packets destined for the broadcast address of a network. A known filter already detects Smurf-like attacks, but ours limits itself to ICMP echo to generate fewer false positives.

Sendmail

This filter looks for attempted and established connections on the SMTP port of machines on the local network whose IP addresses are not enumerated in the filter as legitimate mail servers. This filter thus detects both sendmail port scans and unexpected mail servers; we found such a server on our network. This filter is easily changed to monitor other ports.

Buffer Overflow

This filter looks for a specific pattern in all packet payloads going to a specified port. The current implementation detects a buffer overflow attack against `named` in Linux and FreeBSD. This filter can form the basis for general pattern matching within a packet.

Teardrop

This filter detects the Teardrop attack, looking for overlap in a collection of fragmented TCP packets. The obvious choice of data structure, a list of lists, is not supported by NFR, requiring us to carve up a single array manually. Our solution collects fragmented packets and uses fragment offsets and lengths to check for overlaps; a timeout discards older fragments to limit memory consumption. We collect packets as they arrive and process collected packets periodically; we have not compared this to an implementation that processes each packet as it arrives.

Profiler

We developed a profiling filter to categorize IP traffic. ICMP packets are counted by type; TCP and UDP packets are counted by port. NFR's associative arrays are useful here. This filter gives us a rough characterization of network traffic and helps focus our attention on traffic appropriate to the sifting work. As an immediate result, we observed that CITI has made a lot of progress in replacing `telnet` with `ssh`.

The above scripts were running intermittently against CITI's network traffic over a period of several weeks. Attack programs were used to verify correct filter behavior.

SIFTING SCRIPTS

Here we concentrate on separating suspicious traffic from ordinary traffic. Our strategy is to observe a given stream (say, DNS traffic) for a period of time to note "normal" behavior, say a stream of "A" requests and responses, then write NFR scripts to filter out packets corresponding to this behavior. Playing network traffic through such a filter reduces its volume by suppressing normal traffic to leave "suspicious" traffic behind for manual inspection. This sifting strategy is successful to the extent that normal traffic dominates suspicious packets.

Initially, we based our selection on protocols for which our profiling filter measured high counts. Unfortunately, some of those (`ssh` and `telnet`) defy sifting, and we chose not to inspect others (SMTP) for privacy reasons. Instead, we focused on protocols that have recognizable normal behavior: DNS, FTP, and HTTP. See Appendix B for script listings.

Tcpdump

We wrote an NFR backend that accepts packets from the decision engine and writes them to a log file in `tcpdump` format. This permits other tools, such as `tcpdump` or `tcptrace` [3], to post-process NFR output. It also permits the output of one NFR engine to feed the input of another, which we want to explore further.

We wrote a version of this backend that accepts blocks of packets, which permits N-code scripts to accumulate packet streams before writing them out, in those cases where packets must be accumulated before the decision to log them can be made.

DNS

Using our `tcpdump` backend, we wrote a filter that sifts DNS packets and retains suspicious packets in a log for subsequent inspection. The script inspects A, PTR, and MX requests. It logs packets that contain names that are too long, have too many labels, odd options, or an excessive number of records. DNS requests other than these are also logged.

With this simple filter we are able to remove over 90% of all DNS traffic, sifting out 61,092 of 67,048 packets seen in a testing period conducted 27-28 August 1998.

FTP

This filter examines the FTP control stream for normal commands and begins logging the control and the associated data stream if suspicious commands are seen. Control streams are buffered so that normal commands preceding a suspicious command are logged.

A command found in the stream is first matched with a short list of normal commands, *e.g.* `CDUP` or `PWD`, and discarded if found. Then the arguments and argument lengths are checked, with some combinations flagged as suspicious, such as `RETR /etc/passwd`. Along the way, excessively long commands or those containing control characters are also flagged as suspicious.

With this filter we are able to remove about 25% of the packets from the control stream, sifting out 1,059 of 4,193 packets seen during the two-day period. Our filter does not keep enough state to allow the number of sifted data packets to be calculated. Adding `MACB`† to the list of normal commands improved the sifting rate to 80% in a small sample of 274 control packets.

These results are somewhat disappointing; it seems that the characterization of normal traffic varies from one FTP client to another, so the set of traffic common across clients is very small.

HTTP

A coarse filter for sifting HTTP traffic sifts out all `GET` or `POST` commands and responses with length under 512 bytes. This filter sifts 98% of HTTP traffic, removing 25,301 of 25,889 HTTP packets seen during the two-day period. We did not validate our assumption that packets smaller than the triggering packet length are normal, so this filter is of limited use in its present form.

AFS

We wrote a filter that captures all fragmented IP packets and passes them to our `tcpdump` backend. As a simple application, we used this filter to detect AFS servers sending packets larger than the MTU of the destination host's network.

EXPERIENCES

N-code has many powerful features not found in other scripting languages, such as a list structure and a `foreach` function to walk the elements of a list This is very helpful in looking for one of a group of items in a packet, perhaps comparing a list of IP addresses to the packet's address, or

† `MACB` is used to ask an FTP server to employ the MacBinary protocol when transferring files.

looking for a command in an FTP protocol stream. Lists are also useful to store items captured out of a packet stream. Sparse arrays make a convenient associative function for storing and retrieving items.

N-code has functions for reading arbitrary bytes of a packet, and logic functions for masking and rotating those bytes. This makes it easy to examine fields of protocol headers, even those not preconfigured into NFR. For example, fragment bits of an IP header, not available as a builtin variable in NFR, are easily recovered using `byte` and logical functions.

The NFR architecture separates data selection from backend reporting and processing functions. This makes it possible to generate new backends for novel purposes.

There are some syntactic weaknesses in the language. A ":" not separated by spaces in a timeout filter causes the engine to go into an infinite loop, interpreting this as a request to read the value of a variable from another file every 0 seconds. If the characters "_" or "-" are used in a file name, the functions it contains cannot be referenced from within another file, as these characters collide with the N-code subtraction and hierarchy operators.

Adding a selective branch construct to the N-code language would greatly improve code complexity and readability.

The decision to support implicitly declared variables in the N-code language caused us to expend a considerable amount of debugging time, as unreferenced variables caused by simple misspelling of identifiers are difficult to find. Requiring every variable to be declared would alleviate this problem.

CONCLUSION

We found NFR and N-code to be powerful tools for writing complex scripts for detecting known attacks, and for writing sifters to help us in detecting unknown attacks.

ACKNOWLEDGEMENTS

We thank NFR, Inc. staff members Delores Quade, Dodge Mumford, and Mike Stolarchuk for their assistance in avoiding duplication of effort, interpreting N-code semantics, fixing bugs, and helping us to get started. We also thank Marcus Ranum for appreciating our efforts. This work was partially supported by NFR, Inc.

References

1. Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall, ‘‘Implementing A Generalized Tool For Network Monitoring,’’ pp. 1–8 in *Eleventh Systems Administration Conference (LISA ’97)*, San Diego (October 1997.).
2. Thomas H. Ptacek and Timothy N. Newsham, *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*, Secure Networks, Inc. (January 1998).
3. Shawn Ostermann, `tcptrace`, Ohio University. `http://jarok.cs.ohiou.edu/software/tcptrace`

Appendix A

smurf.nfr

```
#-*-c-*-
#This filter detects packets commonly used in a smurf attack by looking for any
#icmp echo request packet sent to the broadcast address (any address ending in
#255).  This pattern can easily be changed to reflect the broadcast address on
#your network, and this could also be altered to look for all traffic to the
#broadcast address from an outside network.

bcastpat=regcomp("255$");
filter smurf icmp()
{
#Test to see if this packet is an ICMP echo request
  if(icmp.type == 8)
    {
#We are testing to see if the last byte of the address is 255, a common
#broadcast address
      if (regexec(bcastpat,cat(ip.dst)))
     {
       echo ("Possible Smurf attack on ",ip.src, " Using ",
           ip.src, "->" , ip.dst, "0);
     }
    }
}
```

sendmail.nfr

```
#-*-c-*-
#This filter detects attempts to connect to the sendmail port of any machine on your
#network that is not a sendmail server.  Could also easily be adapted to find other
"doorknob rattling."  Put all of the mail servers on you network in this list:
smtpservers=listadd(NULL,141.211.92.141,141.211.92.197,141.211.92.199);
filter badsendmailtcp ip()
{
  if(ip.proto==6)
    {
      if(short(ip.blob,2)==25)
     {
       if((!(ip.dest inside smtpservers))&&(ip.dest inside values:my_network))
         {
           {
          echo("TCP connect to port 25(sendmail) attempted",
               "(not sendmail server) ", ip.src,"->",
               ip.dst,"0);
           }
         }
     }
    }
}
filter badsendmailtcpcon tcp()
{
  if(ip.proto==6)
    {
      if(short(ip.blob,2)==25)
     {
       if(((!(ip.dest inside smtpservers))&&(ip.dest inside values:my_network)))
         {
           if(tcp.conn)
          {
            echo("TCP connect to port 25(sendmail) established",
                 "(not sendmail server) ", ip.src,"->",
                 ip.dst,"0);
          }
           else
          {
            echo("TCP connect to port 25(sendmail) attempted",
                 "(not sendmail server) ", ip.src,"->",
                 ip.dst,"0);
          }
         }
     }
    }
}

filter badsendmailudp ip()
{
  if(ip.proto==17)
    {
```

```
      if(short(ip.blob,2)==25)
     {
       if((!(ip.dest inside smtpservers))&&(ip.dest inside values:my_network))
         {
           echo("UDP send to port 25(sendmail) attempted",
             "(not sendmail server) ", ip.src,"->",ip.dst,"0);
         }
     }
   }
}
```

buffover.nfr

```
#-*-c-*-
#detects a specific named buffer overflow targetd at linux and freebsd platforms
#This filter detects all TCP packets sent to a designated port, then searches them
#for a specific string or attack signature.  We must search for the string,
#since it may be at many offsets within the packet depending on the version of the
#program under attack.  This filter is configured to detect a buffer overflow attack
#on a named which runs under linux and freebsd.  By changing the string definitions
#and the port numbers below, this can be re-configured to detect other attacks.

attack_name_linux="Linux named buffer overflow";
attack_name_freebsd="FreeBSD named buffer overflow";

attack_sig_linux0=blobbytes(0x31,0xc0,0xb0,0x3f,0x31,0xdb,0xb3);
#the 8th byte of the attack may vary, so we must not look for it.
attack_sig_linux1=blobbytes( 0x31,0xc9,0xcd,0x80,0x31,0xc0,
     0xb0,0x3f,0xb1,0x01,0xcd,0x80,0x31,0xc0,0xb0,0x3f,0xb1,0x02,0xcd,0x80,
     0xeb,0x24,0x5e,0x8d,0x1e,0x89,0x5e,0x0b,0x33,0xd2,0x89,0x56,0x07,0x89,
     0x56,0x0f,0xb8,0x1b,0x56,0x34,0x12,0x35,0x10,0x56,0x34,0x12,0x8d,0x4e,
     0x0b,0x8b,0xd1,0xcd,0x80,0x33,0xc0,0x40,0xcd,0x80,0xe8,0xd7,0xff,0xff,
     0xff,"/bin/sh");

attack_sig_freebsd0=blobbytes(0xeb,0x6e,0x5e,0xc6,0x06,0x9a,0x31,0xc9,0x89,0x4e,
     0x01,0xc6,0x46,0x05,0x07,0x88,0x4e,0x06,0x51,0x31,0xdb,0xb3);
#the 23rd byte of the attack may change, so must not look for it.
attack_sig_freebsd1=blobbytes(0x53,
     0x66,0xc7,0x46,0x07,0xeb,0xa7,0x31,0xc0,0xb0,0x5a,0x50,0xeb,0x50,
     0xfe,0xc1,0x51,0x53,0xc6,0x46,0x08,0xb6,0x31,0xc0,0xb0,0x5a,0x50,0xeb,
     0x41,0xfe,0xc1,0x51,0x53,0xc6,0x46,0x08,0xc5,0x31,0xc0,0xb0,0x5a,0x50,
     0xeb,0x32,0xc7,0x46,0x07,0x2f,0x62,0x69,0x6e,0xc7,0x46,0x0b,0x2f,0x73,
     0x68,0x21,0x31,0xc0,0x88,0x46,0x0e,0x8d,0x5e,0x07,0x89,0x5e,0x0f,0x89,
     0x46,0x13,0x8d,0x5e,0x13,0x53,0x8d,0x5e,0x0f,0x53,0x8d,0x5e,0x07,0x53,
     0xb0,0x3b,0x50,0xeb,0x05,0xe8,0x8d,0xff,0xff,0xff);
filter bufferoverflowlinux tcp()
{
  if(tcp.dport==53)
    {
      if( (index(tcp.blob,attack_sig_linux0)!=-1)
     &&(index(tcp.blob,attack_sig_linux1)!= -1)) {
       echo("Found ",attack_name_linux," attack code in TCP connection",ip.src,
            ":",tcp.sourceport,"->",ip.dest, ":",tcp.destport,"0);
     }
    }
}

filter bufferoverflowfreebsd tcp()
{
  if(tcp.dport==53)
    {
      if( (index(tcp.blob,attack_sig_freebsd0)!= -1)
     &&(index(tcp.blob,attack_sig_freebsd1)!= -1)) {
       echo("Found ",attack_name_freebsd," attack code in TCP connection",ip.src,
            ":",tcp.sourceport,"->",ip.dest,":",tcp.destport,"0);
```

```
        }
    }
}
```

teardrop.nfr

```
#-*-c-*-
#This filter detects a teardrop attack by re-assembling a fragmented packet
#stream and checking for overlap of fragments within it.  We keep an array
#containing the source, destination, protocol, id, number of fragments
#collected for that packet and the time a fragment was last received for each
#fragment stream, and the length and offset of each fragment of each stream is
#recorded in another array.  Since the structure of a packet’s information is
#a fixed length, we simply index into the array containing packet stream
#information using an index multiplied by the structure’s length.  We take
#advantage of nfr’s support of a sparse address space to index streams of
#fragments into the fragment array based on their index into the packet array
#left shifted 14 bits.  We multiply the fragment’s index into the array of
#fragments for that fragment stream by the length of the fragment structure
#and OR this with the left shifted index into the packet array to obtain the
#index into the fragment array.  This gives each packet a space for as many
#fragments as are allowed in the offset field of an IP header.  This also
#leaves the potental for an overflow attack if over 2^13 fragemnts of a
#particular stream are directed at the filter.

#Once these fragements are collected, a timeout trigger periodically sorts the
#fragment array, then walks the sorted array and checks adjacent packets for
#overlap.  If overlap is found, it is reported.  After a given pair of
#packets is compared, the first one is marked as having been used in a
#comparison, and if an overlap is found where both packets have already been
#used in a comparison, this is not reported, because it was previously
#reported.  We currently only check linearly and do no checking within groups
#of packets with the same offset, so some attack on the monitor is
#possible here. Of course packets with the same offset overlap, so these are
#reported.   If the packet stream has not received a packet within a given
#time period, it is deleted and the last packet stream in the array is moved
#into its place.

#Obviously this filter has an engine to collect and re-assemble packets, and
#it could easily be modified to check for other anomylies between fragments
#or assemble based on different rules.


pkt_struct_len=6;
frag_struct_len=3;
pkt_count=0;
stale_timeout=120;
last_time=0;

filter teardrop ip()
{

  $inlist=0;

$flags=byte(eth.blob,6)>>5;
$offset=short(eth.blob,6) & 0x1fff;


```

```
#we have a fragmented paket
  if((($flags & 2)!=2)&&(!((($flags & 1)!=1)&&($offset==0))))
    {
     # echo("Fragmented Packet Detected0);
      #display("verbose");
      $tlen=(short(eth.blob,2) );
      $hlen=4*((byte(eth.blob,0))&0x0f);
      $ipid=short(eth.blob,4);
      $offlag=short(eth.blob,6);
      $len=$tlen-$hlen;
      #echo("offlag=",$offlag," len=",$len,"0);
#search packlist for packet
      $current=0;
      $inlist=0;
      while($current<pkt_count)
     {
       $pktoff=$current*pkt_struct_len;
       if((packlist[$pktoff]==$ipid)&&(packlist[$pktoff+1]==ip.src)
          &&(packlist[$pktoff+2]==ip.dst)&&(packlist[$pktoff+3]==ip.proto))
         {
#if found add frag
           $inlist=1;
           #$fragment=fraglist[$current];
           fraglist[($current<<14)+
                (packlist[$pktoff+5]*frag_struct_len)]=$offlag;
           fraglist[($current<<14)+
                ((packlist[$pktoff+5]*frag_struct_len)+1)]=$len;
           fraglist[($current<<14)+
                ((packlist[$pktoff+5]*frag_struct_len)+2)]=0;
           packlist[$pktoff+5]=packlist[$pktoff+5]+1;
           break;
         }
       $current=$current+1;
     }

      if($inlist != 1)
     {
#if not found, add packet
#add new packet
       $pktoff=pkt_count*pkt_struct_len;
       packlist[$pktoff]=$ipid;
       packlist[$pktoff+1]=ip.src;
       packlist[$pktoff+2]=ip.dst;
       packlist[$pktoff+3]=ip.proto;
       packlist[$pktoff+4]=system.time;
       packlist[$pktoff+5]=1;           #;//size of fragment list array
       fraglist[(pkt_count<<14)]=$offlag;
       fraglist[(pkt_count<<14)+1]=$len;
       fraglist[(pkt_count<<14)+2]=0;
       pkt_count=pkt_count+1;
     }
    }

}
```

```


filter checkoverlap timeout(sec : 20, repeat)
{
  #echo("fraglist=",fraglist,"0);

#sort and re-assemble
  $index=0;
  while ($index<pkt_count)
    {
      $k=0;
      while($k<packlist[(($index*pkt_struct_len)+5)])
     {
       $l=0;
       while($l<frag_struct_len)
         {
      #     echo("fraglist[",($index<<14)+($k*frag_struct_len)+$l,"]=",
          #   fraglist[($index<<14)+($k*frag_struct_len)+$l],"0);
           $l=$l+1;
         }
       $k=$k+1;
     }
#sort
      #$frag=fraglist[$index];
      $fragind=$index<<14;
      $size=((packlist[(($index*pkt_struct_len)+5)])*frag_struct_len);
      $current=0;
      while($current<$size)
     {
       $i=$current;
       while($i<$size)
         {
           if((fraglist[$i+$fragind]&0x1fff)<
            (fraglist[$current+$fragind]&0x1fff))
          {
            $j=0;
            while($j<frag_struct_len)
              {
                $temp=fraglist[$i+$j+$fragind];
                fraglist[$i+$j+$fragind]=fraglist[$current+$j+$fragind];
                fraglist[$current+$j+$fragind]=$temp;
                $j=$j+1;
              }
          }
           $i=$i+frag_struct_len;
         }

       $current=$current+frag_struct_len;
     }
#********Do we also have to sort by length?
#walk and check for overlap and duplicates
      $current=frag_struct_len;
      #echo("fraglist=",fraglist,"0);
```

```
     while($current<$size)
    {
      $curoff=(fraglist[$current+$fragind]&0x1fff)*8;
      $prevoff=(fraglist[$current+$fragind-frag_struct_len]&0x1fff)*8;
      $curlen=fraglist[$current+$fragind+1];
      $prevlen=fraglist[$current+$fragind-frag_struct_len+1];
      $curalert=fraglist[$current+$fragind+2];
      $prevalert=fraglist[$current+$fragind-frag_struct_len+2];
    #echo("curoff",$curoff," prevoff",$prevoff," curlen",$curlen,
    #     " prevlen",$prevlen," curalert",$curalert," prevalert",
    #     $prevalert,"0);
      if((($prevoff+$prevlen)>$curoff)
         &&(($curalert==0)||($prevalert==0)))
#overlaping packets print src, dst, both offsets and lengths
        {
          echo(system.time,": Overlaping packet fragments detected ",
            "from ",packlist[($index*pkt_struct_len)+1], " to ",
            packlist[($index*pkt_struct_len)+2]," ID:",
            packlist[$index*pkt_struct_len]," Protocol ",
            packlist[($index*pkt_struct_len)+3],
            " first fragment offset:",
            (8*(fraglist[$current-frag_struct_len+$fragind]&0x1fff)),
            " len:",
            fraglist[$current+1-frag_struct_len+$fragind],
            " second fragment offset:",
            8*(fraglist[$current+$fragind]&0x1fff),
            " len:",fraglist[$current+1+$fragind],"0);
          fraglist[$current+2+$fragind-frag_struct_len]=1;
          if(($current+frag_struct_len)==$size)
         {
           fraglist[$current+2+$fragind]=1;
         }
        }
      $current=$current+frag_struct_len;
    }

     if((packlist[($index*pkt_struct_len)+4])<(system.time-stale_timeout)
     &&((packlist[($index*pkt_struct_len)+4])!=NULL))
    {
      echo("Deleting packet index ",$index ,packlist[($index*pkt_struct_len)+4],"0);


      $pktoff=(pkt_count-1)*pkt_struct_len;
      $newpktoff=$index*pkt_struct_len;
#this will have to move the elements now
#delete elements to be deleted, then move and delete elements moved into
#that space
      $i=0;
      while($i<packlist[$newpktoff+5])
         {
           $j=0;
           while($j<frag_struct_len)
             {
               fraglist[($fragind+($i*frag_struct_len))+$j]=NULL;
```

```
              $j=$j+1;
            }
          $i=$i+1;
        }
      if(pkt_count>1)
        {
          $i=0;
          while($i<packlist[$pktoff+5])
          {
            $j=0;
            while($j<frag_struct_len)
              {
                fraglist[($fragind+($i*frag_struct_len))+$j]
               =fraglist[(((pkt_count-1)<<14)
                        +($i*frag_struct_len))+$j];
                fraglist[(((pkt_count-1)<<14)
                     +($i*frag_struct_len))+$j]=NULL;
                $j=$j+1;
              }
            $i=$i+1;
          }

          packlist[$newpktoff]=packlist[$pktoff];
          packlist[$newpktoff+1]=packlist[$pktoff+1];
          packlist[$newpktoff+2]=packlist[$pktoff+2];
          packlist[$newpktoff+3]=packlist[$pktoff+3];
          packlist[$newpktoff+4]=packlist[$pktoff+4];
          packlist[$newpktoff+5]=packlist[$pktoff+5];
        }
      packlist[$pktoff]=NULL;
      packlist[$pktoff+1]=NULL;
      packlist[$pktoff+2]=NULL;
      packlist[$pktoff+3]=NULL;
      packlist[$pktoff+4]=NULL;
      packlist[$pktoff+5]=NULL;


      pkt_count=pkt_count-1;
    }
    $index=$index+1;
  }

}
```

profile.nfr

```
smtp=0;
ftp=0;
telnet=0;
ssh=0;
http_t=0;
http_u=0;
nntp=0;
ntp_t=0;
ntp_u=0;
ping=0;
dns_u=0;
dns_t=0;
locus_map_t=0;
locus_conn_t=0;
netbios_datagram_t=0;
netbios_name_t=0;
locus_map_u=0;
locus_conn_u=0;
netbios_datagram_u=0;
netbios_name_u=0;
rip=0;
xwin_t=0;
xwin_u=0;
afs_t=0;
afs_u=0;
dec_notes_t=0;
print_spool_t=0;
dec_notes_u=0;
print_spool_u=0;
p1865u=0;
p1046u=0;
kerb_iv_t=0;
netbios_session_t=0;
kerb_iv_u=0;
netbios_session_u=0;
igmp=0;
sql_t=0;
rpcbind_t=0;
sql_t=0;
rpcbind_t=0;
login=0;
whod=0;
pop3_t=0;
pop3_u=0;
unknown_tcp=listadd(NULL,NULL);
unknown_udp=listadd(NULL,NULL);

last_time=0;
last_report=0;

filter countdata ip()
{
```

```
if(ip.proto==6)#;tcp
  {
    $sport=short(ip.blob,0);
    $dport=short(ip.blob,2);
    if(($sport==25)||($dport==25))
   {
     smtp=smtp+1;
     last_time=system.time;
   }
    else
   if(($sport==21)||($dport==21))
     {
       ftp=ftp+1;
       last_time=system.time;
     }
   else
     if(($sport==23)||($dport==23))
       {
         telnet=telnet+1;
         last_time=system.time;
       }
     else
       if(($sport==22)||($dport==22))
         {
        ssh=ssh+1;
        last_time=system.time;
         }
       else
         if(($sport==80)||($dport==80))
        {
          http_t=http_t+1;
          last_time=system.time;
        }
         else
        if(($sport==119)||($dport==119))
          {
            nntp=nntp+1;
            last_time=system.time;
          }
        else
          if(($sport==123)||($dport==123))
            {
              ntp_t=ntp_t+1;
              last_time=system.time;
            }
          else
            if(($sport==53)||($dport==53))
              {
             dns_t=dns_t+1;
             last_time=system.time;
              }
             else
            if(($sport==127)||($dport==127))
              {
```

```
   locus_map_t=locus_map_t+1;
   last_time=system.time;
    }else
 if(($sport==125)‖($dport==125))
    {
   locus_conn_t=locus_conn_t+1;
   last_time=system.time;
    }else
 if(($sport==138)‖($dport==138))
    {
   netbios_datagram_t=netbios_datagram_t+1;
   last_time=system.time;
    }else
 if(($sport==137)‖($dport==137))
    {
   netbios_name_t=netbios_name_t+1;
   last_time=system.time;
    }

 else
 if(($sport==3333)‖($dport==3333))
    {
   dec_notes_t=dec_notes_t+1;
   last_time=system.time;
    }
 else
 if(($sport==515)‖($dport==515))
    {
   print_spool_t=print_spool_t+1;
   last_time=system.time;
    }
 else
 if(($sport==139)‖($dport==139))
    {
   netbios_session_t=netbios_session_t+1;
   last_time=system.time;
    }
 else
 if(($sport==750)‖($dport==750))
    {
   kerb_iv_t=kerb_iv_t+1;
   last_time=system.time;
    }
 else
 if(($sport==1498)‖($dport==1498))
    {
   sql_t=sql_t+1;
   last_time=system.time;
    }             else
 if(($sport==111)‖($dport==111))
    {
   rpcbind_t=rpcbind_t+1;
   last_time=system.time;
    }             else
```

```
            if(($sport==513)‖($dport==513))
              {
             login=login+1;
             last_time=system.time;
              }           else
            if(($sport==110)‖($dport==110))
              {
             pop3_t=pop3_t+1;
             last_time=system.time;
              }
            else
              if((($sport>5999)&&($sport<6064))
              ‖(($dport>5999)&&($dport<6064)))
             {
               xwin_t=xwin_t+1;
               last_time=system.time;
             }
            else
              if((($sport>6999)&&($sport<7010))
              ‖(($dport>6999)&&($dport<7010)))
             {
               afs_t=afs_t+1;
               last_time=system.time;
             }
             else
              {
             if((!($sport inside unknown_tcp))&&($sport<10000))
               {
                 echo ("Unrecognized protocol in TCP header",
                     $sport,"->",$dport," nonce ",
                     packet.nonce,"0);
                 unknown_tcp=listadd(unknown_tcp,$sport);
               }
             if((!($dport inside unknown_tcp))&&($dport<10000))
               {
                 echo ("Unrecognized protocol in TCP header",
                     $sport,"->",$dport," nonce ",
                     packet.nonce,"0);
                 unknown_tcp=listadd(unknown_tcp,$dport);
               }
#display("verbose");
              }
    }
  else
    if(ip.proto==17)#;udp
    {
      $sport=short(ip.blob,0);
      $dport=short(ip.blob,2);
      if(($sport==80)‖($dport==80))
     {
       http_u=http_u+1;
```

```
  last_time=system.time;
}
 else
if(($sport==123)‖($dport==123))
  {
    ntp_u=ntp_u+1;
    last_time=system.time;
  }
else
  if(($sport==53)‖($dport==53))
    {
      dns_u=dns_u+1;
      last_time=system.time;
    }
else
  if(($sport==127)‖($dport==127))
    {
      locus_map_u=locus_map_u+1;
      last_time=system.time;
    }
  else
    if(($sport==125)‖($dport==125))
      {
     locus_conn_u=locus_conn_u+1;
     last_time=system.time;
      }
    else
      if(($sport==138)‖($dport==138))
     {
       netbios_datagram_u=netbios_datagram_u+1;
       last_time=system.time;
     }
      else
     if(($sport==137)‖($dport==137))
       {
         netbios_name_u=netbios_name_u+1;
         last_time=system.time;
       }
     else
       if(($sport==520)‖($dport==520))
         {
           rip=rip+1;
           last_time=system.time;
         }
         else
         if(($sport==3333)‖($dport==3333))
           {
          dec_notes_u=dec_notes_u+1;
          last_time=system.time;
           }
         else
         if(($sport==515)‖($dport==515))
           {
          print_spool_u=print_spool_u+1;
```

```
    last_time=system.time;
     }
  else
  if(($sport==1865)||($dport==1865))
     {
    p1865u=p1865u+1;
    last_time=system.time;
     }
  else
  if(($sport==1046)||($dport==1046))
     {
    p1046u=p1046u+1;
    last_time=system.time;
     }
  else
  if(($sport==139)||($dport==139))
     {
    netbios_session_u=netbios_session_u+1;
    last_time=system.time;
     }
  else
  if(($sport==750)||($dport==750))
     {
    kerb_iv_u=kerb_iv_u+1;
    last_time=system.time;
     }
  else
  if(($sport==1498)||($dport==1498))
     {
    sql_u=sql_u+1;
    last_time=system.time;
     }             else
  if(($sport==111)||($dport==111))
     {
    rpcbind_u=rpcbind_u+1;
    last_time=system.time;
     }             else
  if(($sport==513)||($dport==513))
     {
    whod=whod+1;
    last_time=system.time;
     }             else
  if(($sport==110)||($dport==110))
     {
    pop3_u=pop3_u+1;
    last_time=system.time;
     }
  else
  if((($sport>5999)&&($sport<6064))
      ||(($dport>5999)&&($dport<6064)))
     {
    xwin_u=xwin_u+1;
    last_time=system.time;
     }
```

```
              else
              if((($sport>6999)&&($sport<7010))
                 ||(($dport>6999)&&($dport<7010)))
                {
               afs_u=afs_u+1;
               last_time=system.time;
                }
              else
                {
               if((!($sport inside unknown_udp))&&($sport<10000))
                  {
                    echo ("Unrecognized protocol in UDP header ",
                       $sport,"->",$dport," nonce ",packet.nonce,
                       "0);
                    unknown_udp=listadd(unknown_udp,$sport);
                  }
               if((!($dport inside unknown_udp))&&($dport<10000))
                  {
                    echo ("Unrecognized protocol in UDP header ",
                       $sport,"->",$dport," nonce ",
                       packet.nonce,"0);
#display("verbose");
                    unknown_udp=listadd(unknown_udp,$dport);
                  }
#we need to add these to a list, and only report once
                }
    }
    else
      if(ip.proto==1)#;icmp
    {
      $type=byte(ip.blob,0);
      if(($type=0)||($type=8))
     {
       ping=ping+1;
       last_time=system.time;
     }
      else
     {
       echo ("Unrecognized type in ICMP header ",$type,
          " nonce ",packet.nonce,"0);
       #display("verbose");
     }
    }
      else
     if(ip.proto==2)
       {
         igmp=igmp+1;
         last_time=system.time;
       }
```

```
      else
     {
       echo ("Unrecognized protocol in IP header ",ip.proto,
          "nonce ",packet.nonce,"0);
       #display("verbose");
     }

}

filter output timeout(sec : 120,repeat)
{
  if(last_time>last_report)
    {
      last_report=system.time;
      echo("smtp=",smtp," ftp=",ftp," telnet=",telnet," ssh=",ssh," http_t=",
        http_t," http_u=",http_u," nntp=",nntp," ntp_t=",ntp_t," ntp_u=",
        ntp_u," ping=",ping," dns_t",dns_t," dns_u",dns_u,
        " locus_map_t=",locus_map_t,
        " locus_conn_t=",locus_conn_t,
        " netbios_datagram_t=",netbios_datagram_t,
        " netbios_name_t=",netbios_name_t,
        " locus_map_u=",locus_map_u,
        " locus_conn_u=",locus_conn_u,
        " netbios_datagram_u=",netbios_datagram_u,
        " netbios_name_u=",netbios_name_u," rip=",rip,
        " xwin_t=",xwin_t,
        " xwin_u=",xwin_u,
        " afs_t=",afs_t,
        " afs_u=", afs_u,
        " dec_notes_t=",dec_notes_t,
        " print_spool_t=",print_spool_t,
        " dec_notes_u=",dec_notes_u,
        " print_spool_u=",print_spool_u,
        " p1865u=",p1865u,
        " p1046u=",p1046u,
        " kerb_iv_t=",kerb_iv_t,
        " netbios_session_t=",netbios_session_t,
        " kerb_iv_u=",kerb_iv_u,
        " netbios_session_u=",netbios_session_u,
        " IGMP=",igmp,
        " sql_t=",sql_t,
        " rpcbind_t=",rpcbind_t,
        " sql_t=",sql_t,
        " rpcbind_t=",rpcbind_t,
        " login=",login,
        " whod=",whod,
        " pop3_t=",pop3_t,
        " pop3_u=",pop3_u,


        "0);
    }
}
```

Appendix B

output.c

```
/*
This backend logs packets sent to it one at a time to a tcpdump/pcap format file.  It takes
as argument the file name to write to, and expects input as produced by:

record blobbytes(cat(123456),32,cat(packet.sec),32,cat(packet.usec),32,cat(packet.len),32,
blobbytes(packet.blob)) to test_recorder;
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <pcap.h>
#include <stdlib.h>
#include <limits.h>
#define TCPDUMP_MAGIC 0xa1b2c3d4
int output;
int errlog;
char * signal_sucks;
void cleanup()
{
  close(output);
  close(errlog);
  /*printf("exiting
  exit(0);
}
void crashed()
{
  close(output);
  close(errlog);
  printf("exiting
  abort();
}
main(int argc,char **argv)
{
  char * a,*b;
  int size=1;
  int * data_size;
  int * type;
  int a0;
  char foo;
  struct pcap_file_header hdr;
  struct pcap_pkthdr pkt_hdr;
  signal_sucks=argv[0];
```

```
signal(SIGTERM,(void*)&cleanup);
signal(SIGSEGV,(void*)&crashed);
a=(char*)malloc(1560);
b=a;
data_size=malloc(sizeof(int));
type=malloc(sizeof(int));
hdr.magic = TCPDUMP_MAGIC;
hdr.version_major = PCAP_VERSION_MAJOR;
hdr.version_minor = PCAP_VERSION_MINOR;
/*this should be set to the time zone*/
hdr.thiszone = -4;
hdr.snaplen = 1500;
hdr.sigfigs = 0;
/*this should be set to the real link type*/
hdr.linktype = DLT_EN10MB;

/*This really should do some error checking.*/
if(argc>2)
    {
     printf("Usage
     exit (1);
    }
if(argc==1)
    output=1;
else
    output=open(argv[1],O_WRONLY|O_TRUNC|O_CREAT,0x1ff);
if(output<1)
  {
    printf("Error opening file
    exit(1);
  }
write(output,&hdr,sizeof(hdr));
while (1)
  {
     a0=fread((char*)type,1,4,stdin);
     if(feof(stdin))
       {
         cleanup();
       }
     a0=fread((char*)data_size,1,4,stdin);
     if(feof(stdin))
       {
         cleanup();
       }
    if(*type==1)
   {
     /*   a0=*data_size;
     if(*data_size>1500)
       printf("data_size=%d0,*data_size);
     if(a0!=*data_size)
       {
         printf("*data_size=%x a0=%h",*data_size,a0);
         crashed();
         }*/
```

```
       a=b;
       size=fread(a,1,*data_size,stdin);
       if(feof(stdin))
         {
           cleanup();
         }
       if(strtol(a,&a,10)==123456)
         {
           pkt_hdr.ts.tv_sec=strtol(a,&a,10);
           pkt_hdr.ts.tv_usec=strtol(a,&a,10);
           pkt_hdr.len=strtol(a,&a,10);
           a++;
           pkt_hdr.caplen=pkt_hdr.len;
           /*if(size<(pkt_hdr.len+20))
           {
             printf("Read in file size=%xa[0-3]=%d %d %d %d0,size,a[0],a[1],a[2],a[3]);
          printf("pkt_hdr.ts.tv_sec=%lx pkt_hdr.ts.tv_usec=%lx pkt_hdr.len=%lx 0,pkt_hdr.ts.tv_sec, pkt_hdr.ts.tv_usec, pkt_hdr.len);
           }
           */
           write(output,&pkt_hdr,sizeof(pkt_hdr));
           write(output,a,pkt_hdr.len);
         }
       else
         {
           errlog=open("output_err.log",O_WRONLY|O_CREAT,0x1ff);
           write(errlog,a+7,size-7);
           printf("Read in file size=%x0,size);
           close (errlog);
         }
     }
    }

}
```

encap_out.c

```
/*
This backend logs packets sent to it in blocks to a tcdump/pcap formant file.  It takes
as argument the file name to write to and expects input as produced by:
$data_blob=blobbytes(NULL);
#The next 2 statements are repeated to add packets to the block to be dumped(or discarded.)
This scheme is useful for holding on to data that you might want to log but won't know if
you do until some time after the packets arrive.
$dump_blob=blobbytes(cat(123456),32,cat(packet.sec),32,cat(packet.usec),32,cat(packet.len),
32,blobbytes(packet.blob));
$data_blob=blobbytes($data_blob,cat(strlen($dump_blob)),32,$dump_blob);
record $data_blob to test_recorder;
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <pcap.h>
#include <stdlib.h>
#include <limits.h>
#define TCPDUMP_MAGIC 0xa1b2c3d4
int output;
int errlog;
char * signal_sucks;
void cleanup()
{
  close(output);
  close(errlog);
  /*printf("exiting
  exit(0);
}
void crashed()
{
  close(output);
  close(errlog);
  printf("exiting
  abort();
}
main(int argc,char **argv)
{
  char * a,*b;
  int size=1;
  int * data_size;
  int * type;
  int a0,i,total_read,x,blob_size;
  char foo;
```

```
struct pcap_file_header hdr;
struct pcap_pkthdr pkt_hdr;
signal_sucks=argv[0];
signal(SIGTERM,(void*)&cleanup);
signal(SIGSEGV,(void*)&crashed);
a=(char*)malloc(1560);
b=a;
data_size=malloc(sizeof(int));
type=malloc(sizeof(int));
hdr.magic = TCPDUMP_MAGIC;
hdr.version_major = PCAP_VERSION_MAJOR;
hdr.version_minor = PCAP_VERSION_MINOR;
/*this should be set to the time zone*/
hdr.thiszone = -4;
hdr.snaplen = 1500;
hdr.sigfigs = 0;
/*this should be set to the real link type*/
hdr.linktype = DLT_EN10MB;

/*This really should do some error checking.*/
if(argc>2)
    {
     printf("Usage
     exit (1);
    }
if(argc==1)
    output=1;
else
    output=open(argv[1],O_WRONLY|O_TRUNC|O_CREAT,0x1ff);
if(output<1)
  {
    printf("Error opening file
    exit(1);
  }
write(output,&hdr,sizeof(hdr));
while (1)
  {
     a0=fread((char*)type,1,4,stdin);
     if(feof(stdin))
       {
         cleanup();
       }
     a0=fread((char*)data_size,1,4,stdin);
     if(feof(stdin))
       {
         cleanup();
       }
     total_read=0;
     if(*type==1)
       {
         blob_size=*data_size;
         while(total_read<blob_size)
        {
          a0=0;
```

```
i=0;
a++;
a[-1]=0;
while(a[i-1]!=32)
  {
    x=fread(a+i,1,1,stdin);
    a0+=x;
    total_read+=x;
    if(feof(stdin))
   {
     cleanup();
   }
    i++;
  }
*data_size=strtol(a,NULL,10);

/*   a0=*data_size;
     if(*data_size>1500)
     printf("data_size=%d0,*data_size);
     if(a0!=*data_size)
     {
     printf("*data_size=%x a0=%h",*data_size,a0);
     crashed();
     }*/
a=b;
size=fread(a,1,*data_size,stdin);
total_read+=size;
if(feof(stdin))
  {
    cleanup();
  }
if(strtol(a,&a,10)==123456)
  {
    pkt_hdr.ts.tv_sec=strtol(a,&a,10);
    pkt_hdr.ts.tv_usec=strtol(a,&a,10);
    pkt_hdr.len=strtol(a,&a,10);
    a++;
    pkt_hdr.caplen=pkt_hdr.len;
    /*if(size<(pkt_hdr.len+20))
   {
   printf("Read in file size=%xa[0-3]=%d %d %d %d0,size,a[0],a[1],a[2],a[3]);
   printf("pkt_hdr.ts.tv_sec=%lx pkt_hdr.ts.tv_usec=%lx pkt_hdr.len=%lx 0,pkt_hdr.ts.tv_sec, pkt_hdr.ts.tv_usec, pkt_hdr.len);
   }
    */
    write(output,&pkt_hdr,sizeof(pkt_hdr));
    write(output,a,pkt_hdr.len);
  }
else
  {
    errlog=open("output_err.log",O_WRONLY|O_CREAT,0x1ff);
    write(errlog,a+7,size-7);
    printf("Read in file size=%x0,size);
```

```
                close (errlog);
              }
          }
         }
    }

}
```

alldns.nfr

```
#This is a filter to reject suspected good DNS packets and log all others to the file rawbinary.bin.  It does this with the help of the tcpdump format file writing backend, output.  Requests and replies for A, PTR and MX records are checked for names that are too long, names with too many elements, and odd options or excessive numbers of records and if any of these conditions are found, they are logged.  All other DNS packets are also logged.


#currently removes about 90%(62322 removed, 5385 logged) of the packet stream.  Could do better by applying some rules to eliminate known benign zone transfer and "all" transfer traffic, as well as SOA records.

dns_types=[1,12,15];#,2,6,12,15,16,33];#252,255];
dns_type=[0];
total_qcounts=0;
max_name_char=64;
max_name_labels=8;
pkts_loged=0;
pkts_sifted=0;
tcp_nonce_list=listadd(NULL);

test_schema=library_schema:new(1,["str"],scope());
test_recorder=recorder("bin/output rawbinary.bin","test_schema");
func tcpdump_packet
{
#echo("in tcpdump_packet0);
record blobbytes(cat(123456),32,cat(packet.sec),32,cat(packet.usec),32,cat(packet.len),32,blobbytes(packet.blob))
    to test_recorder;
}

func dns_name
{
####this dose not currenty handle compressed names
#this function takes the offset of a dns format name from the begining of the
#data portion of the ip packet, and returns an array with element 0 a string
#which is the name passed to it, and elemnt 1 being the length of the total
#name in the packet.
$offset=$1;
$y[0]="";
$y[2]=0;
$labels=0;
if($2==6)
     $x=byte(tcp.blob,$offset);
     if($2==17)
     $x=byte(udp.blob,$offset);

#  echo("$x=",$x," $offset=",$offset,"0);
     $offset=$offset+1;
     $y[1]=1;
     while($x>0)
{
 if(($x>max_name_char))
```

```
   {
     $y[2]="label too long";
   }
 $labels=$labels+1;
 if($labels>max_name_labels)
   {
     $y[2]="too many labels";
   }
 if($labels>50)
   {
     break;
   }
  if($2==6)
    $y[0]=cat($y[0],substr(tcp.blob,$offset,$x));

  if($2==17)
    $y[0]=cat($y[0],substr(udp.blob,$offset,$x));
  $y[1]=$y[1]+$x;
  $offset=$offset+$x;
  if($2==6)
    $x=byte(tcp.blob,$offset);
  if($2==17)
    $x=byte(udp.blob,$offset);
  $offset=$offset+1;
  $y[1]=$y[1]+1;
  if($x!=0)
    $y[0]=cat($y[0],".");
}
     return $y;
}

filter dns_tcp tcp(port: 53)
{
# display("verbose");
#we want to check for a query/response size larger than the given query or
#response.  It would involve a lot of overhead to check this on multiple
#packet tcp streams, so we will only check if it is a single packet query
#or response.

  $dns_size=short(tcp.blob,0);
  if(($dns_size>(packet.len-34))&&(packet.len<1500))
    $weird_size=1;
  else
    $weird_size=0;
  $id=short(tcp.blob,0+2);
  $qr=(byte(tcp.blob,2+2)>>7)&0x1;
  $opcode=(byte(tcp.blob,2+2)>>3)&0xf;
  if($opcode==0) $opcode="QUERY(0)";
  if($opcode==1) $opcode="IQUERY(1)";
  if($opcode==2) $opcode="STATUS(2)";
  $aa=(byte(tcp.blob,2+2)>>2)&0x1;
  $tc=(byte(tcp.blob,2+2)>>1)&0x1;
  $rd=(byte(tcp.blob,2+2))&0x1;
  $ra=(byte(tcp.blob,3+2)>>7)&0x1;
```

```
$z=(byte(tcp.blob,3+2)>>4)&0x7;
$rcode=(byte(tcp.blob,3+2))&0xf;
$qdcount=short(tcp.blob,4+2);
$ancount=short(tcp.blob,6+2);
$nscount=short(tcp.blob,8+2);
$arcount=short(tcp.blob,10+2);

if(!(tcp.connhash inside tcp_nonce_list))
  {
    tcp_nonce_list=listadd(tcp_nonce_list,tcp.connhash);
    $a[0]=0;
    $a[1]=system.time;
    $a[2]=0;
    tcp_nonce[tcp.connhash]=$a;
  }
else
  {
    $a=tcp_nonce[tcp.connhash];
    $a[0]=$a[0]+1;
    $a[1]=system.time;
    tcp_nonce[tcp.connhash]=$a;
  }

#we have to make sure this is the first in a stream before trying to decode it
if(($a[0]==0&&$qr==0)||($a[0]==1&&$qr==1))
  {
    $y=dns_name(12+2,6);
    $qtype=short(tcp.blob,12+2+$y[1]);
    $qclass=short(tcp.blob,12+2+$y[1]+2);
    $header=1;
  }
else
  {
    $header=0;
    $y[2]=$a[2];
  }

if((((1)||($qdcount!=1)||($y[2]!=0)||($qclass!=1)||(strcasecmp($opcode,"QUERY(0)"))||($y[1]>256)||$weird_size||$a[2])&&$header)||$a[2])
  {
    #record packet.blob to test_recorder;
    #record "test" to test_recorder;
    #display("verbose");
    tcpdump_packet();
    pkts_loged=pkts_loged+1;
    $a=tcp_nonce[tcp.connhash];
    $a[2]=1;
    tcp_nonce[tcp.connhash]=$a;
    if(header==1) #This assumes the query and first response are in order
   {
      echo("TCP DNS packet ",ip.src,":",tcp.sport,"->",ip.dst,":",tcp.dport," ID=",$id," Q/R=",$qr," Opcode=",$opcode," AA=",$aa," TC=",$tc," RD=",$rd," RA=",$ra," Z=",$z," RCODE=",$rcode," QDCOUNT=",$qdcount," ANCOUNT=",$ancount," NSCOUNT=",$nscount," ARCOUNT=",$arcount," $y[2]=",$y[2]);

```

```
      $qcount=0;
      $total_length=0;
      while($qcount<$qdcount)
        {
#print query contents
          $y=dns_name(12+$total_length+2,6);
          $qname=$y[0];
          $qtype=short(tcp.blob,12+2+$y[1]+$total_length);
          $qclass=short(tcp.blob,12+2+$y[1]+$total_length+2);
          if((!($qtype inside dns_types))||($qdcount!=1)||($y[2]!=0)||($qclass!=1)||(strcasecmp($opcode,"QUERY(0)"))||($y[1]>256))
         echo(" Query ",$qcount," QNAME=",$qname," QTYPE=",$qtype," QCLASS=",$qclass);
          if(total_qcounts[$qtype]==NULL)
         total_qcounts[$qtype]=1;
          else
         total_qcounts[$qtype]=total_qcounts[$qtype]+1;
          $total_length=$total_length+$y[1];
          $qcount=$qcount+1;
        }
      if((!($qtype inside dns_types))||($qdcount!=1)||($y[2]!=0)||($qclass!=1)||(strcasecmp($opcode,"QUERY(0)"))||($y[1]>256))
        echo("0);
    }
     else
    {
      pkts_loged=pkts_loged+1;
    }
   }

  else
   {
     pkts_sifted=pkts_sifted+1;
   }
}



filter dns_udp udp(port: 53)
{
 # echo(packet.sec," ",packet.usec," ",packet.len);
  #record blobbytes(cat(123456),32,cat(packet.sec),32,cat(packet.usec),32,cat(packet.len),32,blobbytes(packet.blob))
   # to test_recorder;

  #tcpdump_packet();
  #display("verbose");
  $id=short(udp.blob,0);
  $qr=(byte(udp.blob,2)>>7)&0x1;
#$qr=(byte(udp.blob,2));
  $opcode=(byte(udp.blob,2)>>3)&0xf;
  if($opcode==0) $opcode="QUERY(0)";
  if($opcode==1) $opcode="IQUERY(1)";
  if($opcode==2) $opcode="STATUS(2)";
  $aa=(byte(udp.blob,2)>>2)&0x1;
```

```
$tc=(byte(udp.blob,2)>>1)&0x1;
$rd=(byte(udp.blob,2))&0x1;
$ra=(byte(udp.blob,3)>>7)&0x1;
$z=(byte(udp.blob,3)>>4)&0x7;
$rcode=(byte(udp.blob,3))&0xf;
$qdcount=short(udp.blob,4);
$ancount=short(udp.blob,6);
$nscount=short(udp.blob,8);
$arcount=short(udp.blob,10);
      $y=dns_name((12),17);
#echo("$y=",$y,"0);
    $qname=$y[0];
    $qtype=short(udp.blob,12+$y[1]);
    $qclass=short(udp.blob,12+$y[1]+2);

    if((!($qtype inside dns_types))||($qdcount!=1)||($y[2]!=0)||($qclass!=1)||(strcasecmp($opcode,"QUERY(0)"))||($y[1]>256)||($tc==1)||($z!=0)||(($rcode!=0)&&($rcode!=3))||($ancount>8)||($nscount>8)||($arcount>8))
 #if((!($qtype inside dns_types))||($qdcount!=1)||($y[2]!=0)||($qclass!=1)||(strcasecmp($opcode,"QUERY(0)"))||($y[1]>256)||($tc==1)||($z!=0)||(($rcode!=0)&&($rcode!=3))||(($ancount>1)&&(($aa==0)||($qtype!=15)))||($ancount>8)||($nscount>8)||($arcount>8))
   {
     #display("verbose");
     tcpdump_packet();
     pkts_loged=pkts_loged+1;
     echo("UDP DNS packet ",ip.src,":",udp.sport,"->",ip.dst,":",udp.dport," ID=",$id," Q/R=",$qr," Opcode=",$opcode," AA=",$aa," TC=",$tc," RD=",$rd," RA=",$ra," Z=",$z," RCODE=",$rcode," QDCOUNT=",$qdcount," ANCOUNT=",$ancount," NSCOUNT=",$nscount," ARCOUNT=",$arcount," $y[2]=",$y[2]);
   }
    else
   {
     pkts_sifted=pkts_sifted+1;
   }
     $qcount=0;
     $total_length=0;
     while($qcount<$qdcount)
       {
#print query contents
         $y=dns_name((12+$total_length),17);
#echo("$y=",$y,"0);
         $qname=$y[0];
         $qtype=short(udp.blob,12+$total_length+$y[1]);
         $qclass=short(udp.blob,12+$total_length+$y[1]+2);
         if((!($qtype inside dns_types))||($qdcount!=1)||($y[2]!=0)||($qclass!=1)||(strcasecmp($opcode,"QUERY(0)"))||($y[1]>256)||($tc==1)||($z!=0)||(($rcode!=0)&&($rcode!=3))||($ancount>8)||($nscount>8)||($arcount>8))
        {
          echo(" Query ",$qcount," QNAME=",$qname," QTYPE=",$qtype," QCLASS=",$qclass);
        }
         if(total_qcounts[$qtype]==NULL)
        total_qcounts[$qtype]=1;
         else
        total_qcounts[$qtype]=total_qcounts[$qtype]+1;
```

```
          $total_length=$total_length+$y[1];
          $qcount=$qcount+1;
        }
      if((!($qtype inside dns_types))||($qdcount!=1)||($y[2]!=0)||($qclass!=1)||(strcasecmp($opcode,"QUERY(0)"))||($y[1]>256)||($tc==1)||($z!=0)||(($rcode!=0)&&($rcode!=3))||($ancount>8)||($nscount>8)||($ancount>8)||($arcount>8))
        {
          echo("0);
        }
}
filter printit timeout(sec : 240,repeat)
{
#echo("total_qcounts=",total_qcounts,"0);
echo("DNS: Packets Sifted=",pkts_sifted," Packets Loged=", pkts_loged,"(",(100*pkts_sifted)/((pkts_sifted)+(pkts_loged)),"% Removed)0);
}
filter garbcoll timeout(sec : 600,repeat)
{
 new_list=listadd(NULL);
  foreach $x inside (tcp_nonce_list)
    {
      $a=tcp_nonce[$x];
      if(($a[1]+300)<system.time)
     {
       tcp_nonce[$x]=NULL;
     }
      else
     {
       new_list=listadd(new_list,$x);
     }
    }
  tcp_nonce_list=new_list;
  new_list=NULL;
}
```

ftp.nfr

```
#This filter discards suspected good control traffic on the FTP port, port 21, and logs all other control packets to the file "ftp.tcpdump" in tcpdump format.  All packets, including all previous control packets on that connection, are logged to this file once a suspicious packet has been detected.  Additionally, the data on the port(s) of the data connection(s) for that control stream are logged for all subsequent packets.  This data is logged through the file "alldns.nfr" to the log file "rawbinary.bin".

#The command is parsed out and checked to see if it is a recognized command. The length of the command is also checked to detect buffer overflow attacks.  The arguments to some commands are checked and the arguments to other commands are assumed to be harmless as long as they aren't too long.  For example, any password is ok, but a user name of root, or a get of passwd is supicious.  When checking arguments, we check for cursor control characters, since these can be used to mask a malicious command.

#NOTE:  This filter assumes that the packet variable tcp.nonce is unique.  This is not guaranteed but "likely" according to the documentation.  If it is a counter, we should be ok.  If it is some sort of hash, we could be counting on uniqueness that does not exist in some cases.

#Currently, we are removing about 40% of the packets from the control stream (690 packets filtered 1052 packets logged 16778 data packets logged.  We are not keeping enough state to calculate the data packets removed).  If we add some rules to sift out pasv commands from netscape ftp streams, we can probably do better.

#We have added a filter to remove the pasv traffic created by netscape, but it does allow a potential attack by a client posing as a netscape client.

#Also removing "MACB" command, now removes 80%(221 packets sifted, 53 packets logged, 0 associated data packets logged).  A bug caused data packets to not be logged).

#we are also removing "quit".

good_commands=listadd(NULL,"LIST","NLST", "QUIT","SYST","CWD","SIZE","MDTM","AUTH","REST","CDUP","PWD","MACB","quit","MKD");#,"TYPE","PORT","USER", "RETR","STOR");
bad_files=listadd(NULL,"passwd","shadow","host");
test_schema=library_schema:new(1,["str"],scope());
test_recorder=recorder("bin/encap_out ftp.tcpdump","test_schema");
#test_recorder=recorder("dd of=ftp.raw","test_schema");
packets_sifted=0;
packets_loged=0;
data_packets_loged=0;

func tcpdump_packet
{
  record blobbytes(cat(123456),32,cat(packet.sec),32,cat(packet.usec),32,cat(packet.len),32, blobbytes(packet.blob))
     to test_recorder;
}
filter ftpclient tcp(client, dport : 21)
{
  declare $suspect_stream inside tcp.connSym;
  declare $data_blob inside tcp.connSym;
  declare $con_pkts inside tcp.connSym;
  declare $pasv inside tcp.connSym;
```

```
declare $mozilla inside tcp.connSym;
$pass=0;
if($con_pkts!=NULL)
  {
    $con_pkts=$con_pkts+1;
  }
else
  {
    $con_pkts=1;
  }
$dump_blob=blobbytes(cat(123456),32,cat(packet.sec),32,cat(packet.usec),32,cat(packet.len)
,32,blobbytes(packet.blob));
$data_blob=blobbytes($data_blob,cat(strlen($dump_blob)),32,$dump_blob);
$cmd_start=index(tcp.blob,cat(0xff));
if($cmd_start!=-1)
  {
#check to see if this is an abort(242) or status?, otherwise flag
#we don't check status
    if(
    strcasecmp(substr(tcp.blob,$cmd_start+1,1),blobbytes(242))‖
    strcasecmp(substr(tcp.blob,$cmd_start+2,1),blobbytes(255)))
   $suspect_stream=1;
  }
else
  {
    $good=0;
#we are checking for cursor control charechters that would let malicious
#arguments otherwise pass through the detector
    foreach $a inside(listadd(NULL,blobbytes(8),blobbytes(3)))#we should also add DEL
   {
     if(index(tcp.blob,$a)!=-1)
       $suspect_stream=1;
   }
    foreach $a inside(good_commands)
   {
     if(index(tcp.blob,$a)!=-1)
       {
         $good=1;
#        if(index($a,"PASS")!=-1)
#       $pass=1;
       }
   }

    if(($good!=1)‖($suspect_stream==1))
   {
#test other commands that require argument testing or processing here
#use substr to find begining of comand, add to get arg, compare this to bad values or add po
rt
     #echo(substr(tcp.blob,0,10),"0);
     if(index(tcp.blob,"TYPE")!=-1)
       {
         $a=index(tcp.blob,"TYPE");
   #    echo ("offset after type",substr(tcp.blob,$a+5,10),"0);
         if((index(substr(tcp.blob,$a+5,1),"I")==0)‖(index(substr(tcp.blob,$a+5,1),"A"
)==0))
```

```
          {
     #    echo ($a,"0);
     #    echo (substr(tcp.blob,$a+5,10),"0);
          $good=1;
          }
         }
       else
         if($a=index(tcp.blob,"PORT")!=-1)
          {
          $arg=substr(tcp.blob,$a+4,tcp.length-$a-4);
          $j=0;
          $n="";
          $i=0;
          while($j<4)
            {

              $b=NULL;
              while((strcasecmp($b,",")!=0)&&($i<15))
                {
               $k=0;
               $b=substr($arg,$i,1);
               if($b!=",")
                 $n=cat($n,$b);
               $i=$i+1;
               $k=$k+1;
                }
              if($k>3)
                {
               $suspect_stream=1;
               $bad_address=1;
               break;
                }
              if($j<3)
                $n=cat($n,".");
              $j=$j+1;
            }
          $address=host($n);
          $j=0;
#         display ("verbose");
          while($j<3)
            {
              $k=0;
              $n=0;
              $b=NULL;
              $c=100;
              while((strcasecmp($b,",")!=0)&&($k<3)&&($c>47))
                {
               $b=substr($arg,$i,1);
               $c=short(blobbytes(0,$b,0,0),0);
               if((strcasecmp($b,",")!=0)&&($c>47))
                 {
                   $n=(10*$n)+($c-48);
                 }
               $i=$i+1;
```

```
               $k=$k+1;
                }
              if($k>3)
                {
               $suspect_stream=1;
               $bad_address=1;
               break;
                }
              if($j<1)
                $port=$n*256;
              else
                $port=$n+$port;
              $j=$j+1;
            }
          if(($suspect_stream==1)&&($bad_address!=1))
            {
#add to list of connections to dump
              bad_ports[cat($address,$port)]=tcp.connhash;
              bad_hash[tcp.connhash]=listadd(bad_hash[tcp.connhash],
                                  cat($address,$port));
            }
          else
            $good=1;
           }
         else
           if($a=index(tcp.blob,"USER")!=-1)
          {
            if((index(substr(tcp.blob,$a+4,tcp.length-$a-4),"root")==-1))
              {
                $good=1;
              }
          }
         else
           if($a=index(tcp.blob,"PASV")!=-1)
          {
            $good=1;
            if($pasv!=0)
              {
                $suspect_stream=1;
              }
            $pasv=1;
          }
         else
           if($a=index(tcp.blob,"PASS")!=-1)
          {
            $good=1;

            if(index(tcp.blob,"mozilla"!=-1))
               {
                 $mozilla=1;
                 $pasv=0;
               }
          }
          else
```

```
          if($a=index(tcp.blob,"RETR")!=-1)
            {
              foreach $b inside(bad_files)
                {
               if((index(substr(tcp.blob,$a+4,tcp.length-$a-4),$b)!=-1))
                 $suspect_file=1;
                }
              if($suspect_file!=1)
                $good=1;
            }
          else
            {
            if($a=index(tcp.blob,"STOR")!=-1)
              {
                foreach $b inside(bad_files)
               {
                 if((index(substr(tcp.blob,$a+4,tcp.length-$a-4),$b)!=-1))
                   $suspect_file=1;
               }
                if($suspect_file!=1)
               $good=1;
              }
            }
     }
   }
     if($good!=1)
     {
       $suspect_stream=1;
     }

#parse stream and compare caommands,
#flag bad commands, and find associated data stream from port command


     if(tcp.length>500)
     {
       $suspect_stream=1;

     }

  if($suspect_stream==1)
    {
     #if($pass!=1)
     #display("verbose");

    }

}
filter ftpserver tcp(server, dport : 21)
{
  declare $suspect_stream inside tcp.connSym;
  declare $data_blob inside tcp.connSym;
  declare $con_pkts inside tcp.connSym;
  declare $pssv inside tcp.connSym;
```

```
declare $mozilla inside tcp.connSym;
if($con_pkts!=NULL)
  {
    $con_pkts=$con_pkts+1;
  }
else
  {
    $con_pkts=1;
  }
$dump_blob=blobbytes(cat(123456),32,cat(packet.sec),32,cat(packet.usec),32,cat(packet.len),32,blobbytes(packet.blob));
$data_blob=blobbytes($data_blob,cat(strlen($dump_blob)),32,$dump_blob);
$resp_start=index(tcp.blob,cat(0xff));
# if($resp_start!=-1)
#  {
 #   if(strcasecmp(substr(tcp.blob,$cmd_start+1,1),blobbytes(242))
#     ||strcasecmp(substr(tcp.blob,$cmd_start+2,1),blobbytes(255)))
#    $suspect_stream=1;
    #check to see if this is part of a break, or status, otherwise flag
#    }
#  else
if(tcp.length>1000)
  {
    $suspect_stream=1;
  }
}
filter embeded_open tcp(noticesession)
{
  declare $data_blob inside tcp.connSym;
  declare $pasv inside tcp.connSym;
  $pasv=5;
  $data_blob=blobbytes(NULL);
}
filter embeded_close tcp(discardsession)
{
declare $suspect_stream inside tcp.connSym;
declare $data_blob inside tcp.connSym;
declare $con_pkts inside tcp.connSym;
declare $pasv inside tcp.connSym;
declare $mozilla inside tcp.connSym;
#note that this test opens up the server to attacks on the anonymous account(or any other account that has been set to take mozilla as a password using the passive command.  This may be an ok compromise, but should be removed for the best security.
if(($pasv==1)&&($mozilla!=1))
  {
    $suspect_stream=1;
  }
if($suspect_stream==1)
  {
```

```
      packets_loged=packets_loged+$con_pkts;
      record $data_blob to test_recorder;
      #echo("$data_blob=",$data_blob,"0);#/*
      #*/ /*#write the commands and responses to a file through a tcpdump output recorder
      #record data_blob*/

    }
  else
    if($con_pkts!=NULL)
      packets_sifted=packets_sifted+$con_pkts;
  foreach $a inside (bad_hash[tcp.connhash])
    {
      bad_ports[$a]=NULL;
    }

  bad_hash[tcp.connhash]=NULL;
}
filter dump_bad_data tcp()
{


  if ((bad_ports[cat(ip.src,tcp.sport)]!=NULL)||
      (bad_ports[cat(ip.dst,tcp.dport)]!=NULL))
    {
      #echo("suspect data packet detected0);
      data_packets_loged=data_packets_loged+1;
      id_alldns:tcpdump_packet();
    }
}

filter printit timeout(sec : 240,repeat)
{
echo("FTP: Packets Sifted=",packets_sifted," Packets Loged=", packets_loged,"(",(100*packets_sifted)/((packets_sifted)+(packets_loged)),"% Removed) Data Packets Loged=",data_packets_loged,"0);

}
```

http.nfr

```
#This filter discards all HTTP traffic of type "get" or "post" that is under a given length (512 bytes in this case).  All other http packets are logged through alldns.nfr.  This is a very coarse filter, in need of refinement.  Perhaps some parsing of the arguments is possible.

good_types=listadd(NULL,GET,POST);
packets_loged=0;
packets_sifted=0;
max_req_len=512;
filter http tcp(client, dport : 80)
{
  good=0;
  bad=0;
  if (tcp.length > max_req_len)
    {
      #echo ("File too large");
      bad=1;
    }
  foreach $type inside (good_types)
    {
      if(index($type,substr(tcp.blob,0,25))!=-1)
     {

       good=1;
     }

    }
  if((good==0)||(bad==1))
    {
#      echo("***************************HTTP Traffic**********************************0);
#      display("verbose");
    id_alldns:tcpdump_packet();
    packets_loged=packets_loged+1;
    }
  else
    {
      packets_sifted=packets_sifted+1;
    }
}
filter printit timeout(sec : 240,repeat)
{
echo("HTTP: Packets Sifted=",packets_sifted," Packets Loged=", packets_loged,"(",(100*packets_sifted)/((packets_sifted)+(packets_loged)),"% Removed)0);
}
```

allfrag.nfr

```
#This filter captures all fragmented IP packets and writes them to the file fragedPkts.pcap using the special backend that writes to a pcap/tcpdump format file, output.  I used this filter to isolate some AFS servers that were sending packets larger than the MTU of the network of the destination host.

test_schema=library_schema:new(1,["str"],scope());
test_recorder=recorder("bin/output fragedPkts.pcap","test_schema");

func tcpdump_packet
{
record blobbytes(cat(123456),32,cat(packet.sec),32,cat(packet.usec),32,cat(packet.len),32,blobbytes(packet.blob))
    to test_recorder;
}
filter allfrags ip()
{
  if(short(eth.blob,6)&0xbfff)
    {
      tcpdump_packet();
#      echo("Begining of fragmented packet0);
#      display("verbose");
#      echo("End of fragmented packet0);

    }

}
```