Global Information Assurance Certification Paper

GIAC Certified Intrusion Analyst (GCIA)
Practical Assignment
Version 3.3 (revised August 19, 2002)

John E Petkovsek

May 30, 2003

Table of Content

Table of Content 2
Summary 4
Part 1 - Describe the state of intrusion detection 5
Introduction 5
Snort Preprocessors 5
The rpcmon preprocessor 6
The Code 8
Testing 18
Part II - Detects 25
Detect #1 Fragmentation of Code Red 25
Source of trace 25
Detect was generated by 26
Probability the source address was spoofed 27
Description of attack 27
Attack mechanism 28
Correlations 29
Evidence of active targeting 30
Severity 30
Defensive recommendation 31
Multiple choice test question 31
Detect #2 RPC portmap request mountd 31
Source of trace 32
Detect was generated by 32
Probability the source address was spoofed 34
Description of attack 34
Attack mechanism 35
Correlations 35
Evidence of active targeting 35
Severity 36
Defensive recommendation 36
Multiple choice test question 36
Detect #3 XMAS scan 37
Source of trace 37
Detect was generated by 37
Probability the source address was spoofed 38
Description of attack 39
Attack mechanism 40
Correlations 42
Evidence of active targeting 42
Severity 42
Defensive recommendation 43
Multiple choice test question 43
Feedback from posting to incidents.org 43
Assignment 3 - "Analyse This" 44
Summary 44
Files Analysed 44

List of Detects 44
Analysis of Top Ten Detects 45
Top Talkers 58
Portscan Analysis 63
Out-Of-Spec Analysis 65
Link Graph 66
Security Recommendations 67
Description of Analysis Process 67
References 68

Summary

This report was written after attending the SANS conference in San Diego during March of 2003 as part of attaining a GIAC Certification as a Certified Intrusion Analyst. It includes a paper on writing Snort preprocessors which details a potential preprocessor for monitoring RPC portmapper requests. It also analyses a fragmented code red attack, an RPC mountd portmap request, and an XMAS scan. It concludes with the analysis of five consecutives days of logs from a university's intrusion detection system.

Part 1 - Describe the state of intrusion detection

Introduction

While looking at an RPC detect in "http://www.incidents.org/logs/Raw" I concluded a new Snort preprocessor would be useful. In many cases RPC services listen to a port that is not a 'well known port'. The clients using the service find the port the service is listening on by consulting the portmapper. The way this works is when an RPC service initializes it registers with the portmapper. When registering it tells the portmapper its program number, version number, and port. The program and version numbers are 'well known numbers'. Clients can then ask the portmapper what port a service is listening on by giving the portmapper the program and version numbers of the service. Some RPC services always listen on the same port but many let the operating system assign a random port. For the later type of services one cannot write a Snort rule to capture packets to/from these services because the port number is not known ahead of time. What's needed is a way to detect port requests to the portmapper and save the port number from the contents of the portmapper reply. Then Snort can log the packets for the service. That's exactly what Snort preprocessors are good at.

Snort Preprocessors

The SANS Track 3 Intrusion Detection In-Depth documentation gives the following description of Snort preprocessors:

"Preprocessors were introduced in version 1.5 of Snort. They allow the functionality of Snort to be extended by allowing users and programmers to drop modular "plugins" into Snort fairly easily. Preprocessor code is run before the detection engine is called, but after the packet has been decoded. The packet can be modified or analysed in an 'out of band' manner through this mechanism."

The above documentation also lists the preprocessors that ship with Snort:

- http_decode: normalize web traffic for signature analysis
- portscan: detect portscans
- frag2: perform IP defragmentation
- stream4: perform TCP stream reassembly
- arp_spoof: detect hostile ARP activity
- bo: Back Orifice detection
- telnet_decode: telnet negotiation code normalization
- rpc_decode: rpc normalization

As can be deduced from the above list most of these preprocessors work by examining multiple packets (to detect scans, reassemble fragmented packets, ect.). This is what an RPC monitor would have to do if it is to relate portmapper replies to future RPC service packets. The RPC decode preprocessor that ships with Snort performs a useful but different function than the RPC monitor I saw a need for. It (rpc_decode) combines packets that make up an RPC message to the portmapper

request so someone cannot evade Snort's detection engine by splitting the request into several packets.

The source code for Snort preprocessors is found in BASE/src/preprocessors. If a new source file is added then the makefile in BASE/src/preprocessors/Makefile needs to be modified to build Snort with the new preprocessor. In my case the rpc_mon code was added to the following lines as shown:

```
libspp_a_SOURCES = spp_arpspoof.c spp_arpspoof.h spp_bo.c spp_bo.h \
spp_frag2.c spp_frag2.h spp_http_decode.c spp_http_decode.h \
spp_portscan.c spp_portscan.h spp_rpc_decode.c spp_rpc_decode.h  \
spp_stream4.c spp_stream4.h spp_telnet_negotiation.c \
spp_telnet_negotiation.h spp_asn1.c spp_asn1.h spp_fnord.c spp_fnord.h \
spp_conversation.c spp_conversation.h spp_portscan2.c spp_portscan2.h \
spp_perfmonitor.c spp_perfmonitor.h \
spp_rpc_mon.c spp_rpc_mon.h

libspp_a_OBJECTS =  spp_arpspoof.o spp_bo.o spp_frag2.o \
spp_http_decode.o spp_portscan.o spp_rpc_decode.o spp_stream4.o \
spp_telnet_negotiation.o spp_asn1.o spp_fnord.o spp_conversation.o \
spp_portscan2.o spp_perfmonitor.o spp_rpc_mon.o
```

This will get a preprocessor compiled and linked into the Snort binary when the main makefile in BASE/src is executed. Before this is done however the hooks must be put into Snort to get it to call the new preprocessor routines. This is done by modifying the InitPreprocessors() function in BASE/src/plugbase.c. Each preprocessor needs to have a Setup function and InitPreprocessors() will call that function. The Setup function then calls Snorts RegisterPreprocessor function.

Once the preprocessor has been built into the Snort binary the Snort configuration file must be updated to tell Snort to use the new preprocessor. This is done using the preprocessor keyword as follows:

```
preprocessor <name>: <options>
```

The rpcmon preprocessor

I modelled rpcmon after the other Snort preprocessors for consistency. It has the following functions:

```
void SetupRPCmon()
void RPCmonInit(u_char *args)
void ParseRPCmonArgs(char *args)
void RPCmonPreprocFunction(Packet *p)
void RPCmonCleanExitFunction(int signal)
int AddRPCmonEntry(RPCmonList *list, char *entry)
void DeleteRPCmonEntry(RPCmonList *list, RPCmonNode *node)
void FreeRPCmonList(RPCmonList *rpcmon_list)
RPCmonNode *FindRPCmonReq(Packet *p)
RPCmonNode *FindRPCmonPort(Packet* p)
```

The SetupRPCmon routine is what registers the preprocessor with Snort as mentioned earlier. When registering it passes Snort its name "rpcmon" and a function to call during initialization (RPCmonInit).

The RPCmonInit routine parses command line arguments, registers the rpcmon exit function (RPCmonCleanExitFunction) with Snort and creates two linked lists. One list (the request list) is used to hold requests to the portmapper. Since portmapper replies do not contain the request information (program number, etc.) it's necessary to keep this information in order to make sense of the reply. The second linked list (the port list) is used to hold information related to the RPC service port being accessed.
An entry in the request list contains the following information:
- The source address of the requestor
- The source port of the requestor
- The address of the server being accessed
- The port of the server being accessed
- The transaction ID of the request
- The time this entry will expire

An entry in the port list contains:
- The server address
- The server port (port of the RPC service to be monitored)
- The number of packets sent to/from this server port since the entry was created
- The time this entry will expire

The ParseRPCmonArgs routine looks for options specified in the snort.conf file. The optional parameters for rpcmon are:
- the program number to monitor
- the maximum number of packets to log for this service
- the maximum time packets will be logged for this service

Each of these parameters is specified using a keyword followed by a value. An example rpcmon entry in snort.conf would be:

```
preprocessor rpcmon: program 1005 packets 2 timeout 30
```

The RPCmonCleanExitFunction routine is called by Snort when exiting. It frees the memory associated with the linked lists.

The AddRPCmonEntry routine is used to add an entry to one of the linked lists. Its inputs are a pointer to the list and a pointer to the entry to be added.

The DeleteRPCmonEntry routine deletes an entry from one of the linked lists. Its inputs are a pointer to the list and a pointer to the node to be deleted.

The FreeRPCmonList routine is called by RPCmonCleanExitFunction and frees the memory associated with each entry in a list.

The FindRPCmonReq routine finds an entry in the request list. Its input is a pointer to the packet being processed. It goes through the list looking for an entry that matches the packet.

A match is when the stored requests source address/port, destination address/port, and RPC transaction ID matches that of the packet. The FindRPCmonReq routine also deletes expired request list entries.

The FindRPCmonPort routine finds an entry in the port list. Its input is a pointer to the packet being processed. It goes through the list looking for an entry that matches the packet. A match is when the server address/port matches the source or destination address/port of the packet. The FindRPCmonPort routine also deletes expired port list entries.

The RPCmonPreprocFunction routine is where most of the work is done for rpcmon. It is called by Snort whenever a packet is received. Snort passes this function a pointer to the packet information. The structure for this input parameter is named 'Packet' and can be found in BASE/src/decode.h. It contains pointers to the ethernet header, IP header, etc. If a packet doesn't contain a particular header then that pointer will be NULL.

The RPCmonPreprocFunction routine first verifies that a packets IP header and data pointers are not NULL. It then looks at the source and destination port fields in the Packet structure. These will be 0 if the packet isn't a TCP or UDP packet. If the destination port is 111 (the portmapper) and the RPC portion of the packet indicates it is a portmapper request then selected information from the packet is saved in the request list. This information includes the source and destination addresses and ports and the RPC transaction ID. If the source port is 111 then the RPCmonPreprocFunction calls FindRPCmonReq to see if a matching request had previously been saved. If so it gets the port for the RPC service from this portmapper reply and creates an entry in the port list. This entry will contain the services address and port. The RPCmonPreprocFunction routine also checks to see if the packet matches an existing entry in the port list. If so then it is a packet to/from the service we are interested in and the packet is logged to the alert file. To log a packet to the alert file the Snort routines SetEvent and CallAlertFuncs are called.

The Code

Below is the finished code:

```
/*
*
* Snort Preprocessor Plugin
*
* Author: John Petkovsek
*
* Purpose:
*
* This preprocessor looks for portmapper responses to RPC Get Port requests
* for a particular program number, saves the port assigned by the
* portmapper, and then logs packets to/from that port.
*
* Arguments:
```

```
*
* The RPC program number, the maximum number of packets to log, and the
maximum
* time (in seconds) to spend monitoring any particular port.
* i.e. preprocessor rpcmon: program 10005 packets 50 timeout 600
*
* Note: this preprocessor has only been tested on Solaris systems.
*
*/

#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <rpc/rpc.h>
#include <rpc/rpc_msg.h>

#include "generators.h"
#include "log.h"
#include "detect.h"
#include "decode.h"
#include "event.h"
#include "plugbase.h"
#include "parser.h"
#include "mstring.h"
#include "debug.h"
#include "util.h"


#define MODNAME "spp_rpc_monitor"

typedef struct _RPCmonNode
{
    char   *entry;
    struct _RPCmonNode *prev;
    struct _RPCmonNode *next;
} RPCmonNode;

typedef struct
{
    RPCmonNode *head;
} RPCmonList;

typedef struct _RPCmonReqEntry
{
    u_int srcAddr;
```

```
    u_int srcPort;
    u_int destAddr;
    u_int destPort;
    u_int rpcXID;
    u_int endTime;
} RPCmonReqEntry;

typedef struct _RPCmonPortEntry
{
    u_int svrAddr;
    u_int svrPort;
    u_int pktCount;
    u_int endTime;
} RPCmonPortEntry;


/* list of function prototypes for this preprocessor */
void RPCmonInit(u_char *);
void ParseRPCmonArgs(char *);
void RPCmonPreprocFunction(Packet *);
void RPCmonCleanExitFunction(int);
void FreeRPCmonList(RPCmonList *rpcmon_list);
void DeleteRPCmonEntry(RPCmonList *list, RPCmonNode *node);
int  AddRPCmonEntry(RPCmonList *list, char *entry);
RPCmonNode *FindRPCmonPort(Packet *p);
RPCmonNode *FindRPCmonReq(Packet *p);
void PrintRPCmonReqList();
void PrintRPCmonPortList();

typedef struct
{
    u_int address;
    u_int program;
    u_int version;
    u_int protocol;
} GET_PORT_ARGS;

typedef struct
{
    u_int port;
} GET_PORT_RESP;


/* globals */
RPCmonList *rpcmon_portlist;
RPCmonList *rpcmon_reqlist;
u_int      rpcmon_progNum;
u_int      rpcmon_maxPkts;
u_int      rpcmon_timeout;
```

```
/* external globals from rules.c */
extern char *file_name;
extern int file_line;

void SetupRPCmon()
{
   RegisterPreprocessor("rpcmon", RPCmonInit);

   DEBUG_WRAP(DebugMessage(DEBUG_INIT,
         "MODNAME is setup...\n"););
}

void RPCmonInit(u_char *args)
{
   DEBUG_WRAP(DebugMessage(DEBUG_INIT,
         "MODNAME Initialized\n"););

   /* set global parms to their defaults */
   rpcmon_progNum = 0;
   rpcmon_maxPkts = 50;
   rpcmon_timeout = 300;

   /* parse the argument list from the rules file */
   ParseRPCmonArgs(args);
   AddFuncToPreprocList(RPCmonPreprocFunction);

   /* Set the preprocessor function into the function list
   AddFuncToCleanExitList(RPCmonCleanExitFunction);

   /* create request list and port list */
   rpcmon_reqlist = (RPCmonList *)malloc(sizeof(RPCmonList));
   memset(rpcmon_reqlist, 0, sizeof(RPCmonList));
   rpcmon_portlist = (RPCmonList *)malloc(sizeof(RPCmonList));
   memset(rpcmon_portlist, 0, sizeof(RPCmonList));
}

void ParseRPCmonArgs(char *args)
{
   char **Tokens=0;
   int   iTokenNum=0;
   int   ii;

   if (!args)
      return;

   if( args )
   {
     Tokens = mSplit(args, " ", 10, &iTokenNum, '\\');
   }
```

```
   if (iTokenNum < 2)
   {
     FatalError(
       "RPCmon: Invalid arguments - %s (sample usage: preprocessor rpcmon:
program 10005) \n",
       args);
   }

   LogMessage("rpcmon arguments:\n");

   for( ii = 0; ii < iTokenNum; ii++ )
   {
      /* Check for a 'program number' parameter */
      if( strcmp( Tokens[ii],"program")==0 )
      {
         /* make sure we have at least one more argument */
         if( ii == (iTokenNum-1) )
         {
            FatalError("%s(%d) => Invalid Program Number.  The value must be a "
                  "positive integer number.\n", file_name, file_line);
         }
         LogMessage("    Program Number: %s\n", Tokens[ii+1]);
         rpcmon_progNum =  atoi(Tokens[ii+1]);
         if(!rpcmon_progNum)
         {
            FatalError("%s(%d) => Invalid Program Number. The value must be a "
                  "positive integer number.\n", file_name, file_line);
         }

         ii++;
      }
      /* Check for a 'packets' parameter */
      else if( strcmp( Tokens[ii],"packets")==0 )
      {
         /* make sure we have at least one more argument */
         if( ii == (iTokenNum-1) )
         {
            FatalError("%s(%d) => Invalid Max Packets.  The value must be a "
                  "positive integer number.\n", file_name, file_line);
         }
         LogMessage("    Max Packets: %s\n", Tokens[ii+1]);
         rpcmon_maxPkts =  atoi(Tokens[ii+1]);
         if(!rpcmon_maxPkts)
         {
            FatalError("%s(%d) => Invalid Max Packets. The value must be a "
                  "positive integer number.\n", file_name, file_line);
         }

         ii++;
      }
```

```
        /* Check for a 'timeout' parameter */
        else if( strcmp( Tokens[ii],"timeout")==0 )
        {
            /* make sure we have at least one more argument */
            if( ii == (iTokenNum-1) )
            {
                FatalError("%s(%d) => Invalid timeout.  The value must be a "
                        "positive integer number.\n", file_name, file_line);
            }
            LogMessage("    Timeout: %s\n", Tokens[ii+1]);
            rpcmon_timeout =  atoi(Tokens[ii+1]);
            if(!rpcmon_timeout)
            {
                FatalError("%s(%d) => Invalid timeout. The value must be a "
                        "positive integer number.\n", file_name, file_line);
            }

            ii++;
        }
        else
        {
            FatalError("%s(%d)=> Invalid parameter '%s' to preprocessor"
                    " PerfMonitor.\n", file_name, file_line, Tokens[ii]);
        }
    }

    return;
}


void RPCmonPreprocFunction(Packet *p)
{
    struct rpc_msg *rpcPtr;
    GET_PORT_ARGS *args;
    GET_PORT_RESP *resp;
    RPCmonNode     *node;
    RPCmonReqEntry  *reqEntry;
    RPCmonPortEntry *portEntry;
    Event event;

    if (p != NULL)
    {
      if (p->data != NULL && p->iph != NULL)
      {
        if (p->dp == 111)   /* if portmapper request */
        {
          rpcPtr = (struct rpc_msg*)p->data;
          if (rpcPtr->rm_direction == CALL)
          {
            reqEntry = (RPCmonReqEntry*)malloc(sizeof(RPCmonReqEntry));
```

```
          reqEntry->srcAddr = p->iph->ip_src.s_addr;
          reqEntry->destAddr = p->iph->ip_dst.s_addr;
          reqEntry->srcPort = p->sp;
          reqEntry->destPort = p->dp;
          reqEntry->rpcXID = rpcPtr->rm_xid;
          reqEntry->endTime = p->pkth->ts.tv_sec + rpcmon_timeout;
          args = (GET_PORT_ARGS*)&rpcPtr->ru.RM_cmb.cb_verf;
          if (args->program == rpcmon_progNum)
          {
            AddRPCmonEntry(rpcmon_reqlist, (char*)reqEntry);
          }
        }
      }
      if (p->sp == 111)  /* portmapper reply */
      {
        node = FindRPCmonReq(p);
        if (node != NULL)
        {
          DeleteRPCmonEntry(rpcmon_reqlist, node);
          rpcPtr = (struct rpc_msg*)p->data;
          if (rpcPtr->rm_direction == REPLY && rpcPtr->ru.RM_rmb.rp_stat ==
MSG_ACCEPTED)
          {
            portEntry = (RPCmonPortEntry*)malloc(sizeof(RPCmonPortEntry));
            portEntry->svrAddr = p->iph->ip_src.s_addr;
            resp = (GET_PORT_RESP*)&rpcPtr->ru.RM_rmb.ru.RP_ar.ar_stat;
            portEntry->svrPort = resp->port;
            portEntry->pktCount = 0;
            portEntry->endTime = p->pkth->ts.tv_sec + rpcmon_timeout;
            AddRPCmonEntry(rpcmon_portlist, (char*)portEntry);
          }
        }
      }
      node = FindRPCmonPort(p);    /* packet to/from RPC service */
      if (node != NULL)
      {
        portEntry = (RPCmonPortEntry*)node->entry;
        portEntry->pktCount += 1;
        SetEvent(&event, GENERATOR_SPP_ARPSPOOF,
                RPCMON_SERVICE_PACKET, 1, 0, 0, 0);

        CallAlertFuncs(p, RPCMON_SERVICE_PACKET_STR,
                NULL, &event);

        DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN,
                "MODNAME: Packet detected for RPC service \n"););
      }
    }
  }
  else
```

```
    {
        /* p, eh or ah was NULL, either way it's a non-happy packet */
        return;
    }
}

void RPCmonCleanExitFunction(int signal)
{
    FreeRPCmonList(rpcmon_reqlist);
    free(rpcmon_reqlist);
    rpcmon_reqlist = NULL;

    FreeRPCmonList(rpcmon_portlist);
    free(rpcmon_portlist);
    rpcmon_portlist = NULL;
}


int AddRPCmonEntry(RPCmonList *list, char *entry)
{
    RPCmonNode *node;

    node = (RPCmonNode*)malloc(sizeof(RPCmonNode));
    node->entry = entry;

    if (list == NULL)
        return 1;

    node->prev = NULL;
    node->next = list->head;
    list->head = node;
    if (node->next != NULL)
      node->next->prev = node;

    return 0;
}

void DeleteRPCmonEntry(RPCmonList *list, RPCmonNode *node)
{
    if (node->prev == NULL && node->next == NULL)
    {
        list->head = NULL;
    }
    else if (node->prev == NULL)
    {
        list->head = node->next;
        node->next->prev = NULL;
    }
    else if (node->next == NULL)
    {
```

```
        node->prev->next = NULL;
    }
    else
    {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }
    free(node->entry);
    free(node);
}

void FreeRPCmonList(RPCmonList *rpcmon_list)
{
    RPCmonNode *prev;
    RPCmonNode *current;

    if (rpcmon_list == NULL)
        return;

    current = rpcmon_list->head;
    while (current != NULL)
    {
        prev = current;
        current = current->next;
        free(prev->entry);
        free(prev);
    }
    rpcmon_list->head = NULL;
}


RPCmonNode *FindRPCmonReq(Packet *p)
{
    RPCmonNode *current, *next;
    u_int      xid;
    struct rpc_msg *rpcPtr;
    RPCmonReqEntry *entry;

    if (rpcmon_reqlist == NULL)
        return NULL;

    rpcPtr = (struct rpc_msg*)p->data;
    xid = rpcPtr->rm_xid;
    current = rpcmon_reqlist->head;
    while (current != NULL)
    {
        entry = (RPCmonReqEntry*)current->entry;

        if (p->pkth->ts.tv_sec >= entry->endTime)
        {
```

```
            next = current->next;
            DeleteRPCmonEntry(rpcmon_reqlist, current);
            current = next;
            continue;
        }

        if (entry->srcAddr == p->iph->ip_dst.s_addr &&
            entry->srcPort == p->dp &&
            entry->destAddr == p->iph->ip_src.s_addr &&
            entry->destPort == p->sp &&
            entry->rpcXID == xid)
        {
            DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN,
                "MODNAME: FindRPCmonReq() match!"););

            return current;
        }
        current = current->next;
    }
    return NULL;
}

RPCmonNode *FindRPCmonPort(Packet* p)
{
    RPCmonNode *current, *next;
    RPCmonPortEntry *entry;

    if (rpcmon_portlist == NULL)
        return NULL;

    current = rpcmon_portlist->head;
    while (current != NULL)
    {
        entry = (RPCmonPortEntry*)current->entry;

        /* see if entry has expired */
        if (p->pkth->ts.tv_sec >= entry->endTime || entry->pktCount >=
rpcmon_maxPkts)
        {
            next = current->next;
            DeleteRPCmonEntry(rpcmon_portlist, current);
            current = next;
            continue;
        }

        if ( (entry->svrAddr == p->iph->ip_src.s_addr && entry->svrPort == p->sp) ||
             (entry->svrAddr == p->iph->ip_dst.s_addr && entry->svrPort == p->dp) )
        {
            DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN,
                "MODNAME: FindRPCmonPort() match!"););
```

```
            return current;
        }
        current = current->next;
    }
    return NULL;
}
```

Testing

Once I wrote the preprocessor I needed to test it. To do so I wrote an RPC client and server. The code for these is included below.

```
/*
*
* RPC client
*
* Author: John Petkovsek
*
* Purpose:
*
* Send data to an RPC service and receive the response.
*
* Input Parameters:
*
* The hostname of the server, the RPC program number of the service, and the
number of calls
* to make to the service.
*
*/

#include "netdb.h"
#include "msg.h"
#include "errno.h"

main(argc,argv)
int argc;
char **argv;
{
  int    prog_num, sock;
  struct sockaddr_in  svr_addr;
  struct hostent      *hp;
  u_long dest_ip_addr;
  CLIENT *rpc_clt;
  MSG    msg;
  MSG    status;
  enum   clnt_stat  clnt_stat;
  bool_t (*xdr_func)();
```

```
  struct timeval timeout;
  u_int count, ii;

  if (argc != 4)
  {
    printf("usage: %s hostname prog_num num_xfers \n", argv[0]);
    exit(0);
  }
  prog_num = atoi(argv[2]);
  count = atoi(argv[3]);

  /* initialize rpc */

  /* Fill in sockaddr_in structure with address of host we're sending to */
  bzero((char*)&svr_addr, sizeof(svr_addr));
  svr_addr.sin_family = AF_INET;
  if ((hp=gethostbyname(argv[1])) == 0)
  {
    printf("ERROR: gethostbyname failed for %s, errno=%d \n", argv[1], errno);
    exit(0);
  }
  bcopy(hp->h_addr, (char*)&dest_ip_addr, hp->h_length);
  svr_addr.sin_addr.s_addr = dest_ip_addr;
  svr_addr.sin_port = 0;

  sock = RPC_ANYSOCK;
  if ((rpc_clt = (CLIENT*)clntudp_create(&svr_addr, prog_num, 1, timeout, &sock)) ==
NULL)
  {
    printf (" clnt_create failed, errno = %d \n", errno);
    exit(0);
  }

  /* set the retry timeout */
  timeout.tv_sec = 2;
  timeout.tv_usec = 0;
  if (!clnt_control(rpc_clt, CLSET_RETRY_TIMEOUT, (char*)&timeout))
  {
    printf("clnt_control failed, errno = %d \n", errno);
    exit(0);
  }

  xdr_func = xdr_MSG;
  msg.val = 1;
  for (ii=0; ii<count; ii++)
  {
    clnt_stat = clnt_call(rpc_clt, 1, xdr_func, (const  caddr_t)&msg, xdr_func,
                (const  caddr_t)&status, timeout);
    if (clnt_stat != RPC_SUCCESS)
    {
```

```
      printf("clnt_call failed, errno = %d \n", errno);
      exit(0);
    }
  }

  printf("status.val = %d \n", status.val);

  /* free args */
  if (!clnt_freeres(rpc_clt, xdr_func, (caddr_t)&status))
  {
    printf ("clnt_freeres failed, errno = %d \n", errno);
    exit(0);
  }

}


/*
*
* RPC server
*
* Author: John Petkovsek
*
* Purpose:
*
* Register an RPC service with the portmapper (registering an ephemeral port) and
send a
* response when the service is called.
*
* Input Parameters:
*
* The program number of the service
*
*/

#include "msg.h"
#include "errno.h"

main(argc,argv)
int argc;
char **argv;
{
  int     prog_num;
  SVCXPRT *transp;
  static  void service_1();

  if (argc != 2)
  {
    printf("usage: rpc_recv prog_num \n");
    exit(1);
```

```
  }
  prog_num = atoi(argv[1]);

  /* initialize rpc */

  (void)pmap_unset(prog_num, 1);

  transp = (SVCXPRT*)svcudp_create(RPC_ANYSOCK);
  if (transp == NULL)
  {
    printf("svcudp_create failed, errno = %d \n", errno);
    exit(0);
  }

  if (!svc_register(transp, prog_num, 1, service_1, IPPROTO_UDP))
  {
    printf("svc_register failed, errno = %d \n", errno);
    exit(0);
  }

  svc_run();  /* should never return */

  exit(0);
}

static void
service_1(struct svc_req *rqstp, SVCXPRT *transp)
{
  bool_t   (*xdr_func)();
  MSG    msg;
  MSG    status;

  printf("service called \n");

  xdr_func = xdr_MSG;

  bzero((char*)&msg, sizeof(msg));
  if (!svc_getargs(transp, xdr_func, (caddr_t)&msg))
  {
    printf ("svc_getargs failed, errno = %d \n", errno);
    exit(0);
  }

  printf ("msg.val = %d \n", msg.val);

  /* send status */
  status.val = 2;
  if (!svc_sendreply(transp, xdr_func, (const caddr_t)&status))
  {
    printf ("svc_sendreply failed, errno = %d \n", errno);
```

```
    exit(0);
  }

  /* free args */
  if (!svc_freeargs(transp, xdr_func, (caddr_t)&msg))
  {
    printf ("svc_freeargs failed, errno = %d \n", errno);
    exit(0);
  }
}
```

RPC uses XDR encoding for the data passed to/from a service so I also created an XDR structure in a file called msg.x and used rpcgen to build msg_xdr.h and msg_xdr.c.

msg.x:

```
struct MSG {
  int val;
};
```

To compile the client and server I used the following commands:

```
gcc -o rpc_send -lnsl msg_xdr.c rpc_send.c
gcc -o rpc_recv -lnsl msg_xdr.c rpc_recv.c
```

The packets transferred during a test run were as follows:

```
client -> server    PORTMAP C GETPORT prog=1005 (?) vers=1 proto=UDP
server -> client    PORTMAP R GETPORT port=700
client -> server    RPC C XID=1050952157 PROG=1005 (?) VERS=1 PROC=1
server -> client    RPC R (#3) XID=1050952157 Success
```

And Snort logged the following to the alert file:

```
[**] [112:1:1] (spp_rpcmon) Packet detected for RPC service [**]
04/26-15:02:29.238648 111.111.111.111:767 -> 222.222.222.222:700
UDP TTL:255 TOS:0x0 ID:54756 IpLen:20 DgmLen:72 DF
Len: 52

[**] [112:1:1] (spp_rpcmon) Packet detected for RPC service [**]
04/26-15:02:29.238945 222.222.222.222:700 -> 111.111.111.111:767
UDP TTL:255 TOS:0x0 ID:43054 IpLen:20 DgmLen:56 DF
Len: 36
```

Part II - Detects

Detect #1 Fragmentation of Code Red

19:34:12.036507 0:3:e3:d9:26:c0 0:0:c:4:b2:33 0800 1482: 213.107.222.172.4114 > 115.74.227.206.80: P [bad tcp cksum 7592!] 1296348126:1296349554(1428) ack 1419347518 win 17520 (frag 42741:1448@0+) (ttl 111, len 1468, bad cksum c4a5!)

19:34:17.826507 0:3:e3:d9:26:c0 0:0:c:4:b2:33 0800 1482: 213.107.222.172.4114 > 115.74.227.206.80: P [bad tcp cksum 7592!] 0:1428(1428) ack 1 win 17520 (frag 43027:1448@0+) (ttl 111, len 1468, bad cksum c387!)

19:34:29.306507 0:3:e3:d9:26:c0 0:0:c:4:b2:33 0800 1482: 213.107.222.172.4114 > 115.74.227.206.80: P [bad tcp cksum 7592!] 0:1428(1428) ack 1 win 17520 (frag 43550:1448@0+) (ttl 111, len 1468, bad cksum c17c!)

19:34:52.586507 0:3:e3:d9:26:c0 0:0:c:4:b2:33 0800 1482: 213.107.222.172.4114 > 115.74.227.206.80: P [bad tcp cksum 7592!] 0:1428(1428) ack 1 win 17520 (frag 44682:1448@0+) (ttl 111, len 1468, bad cksum bd10!)

19:35:38.946507 0:3:e3:d9:26:c0 0:0:c:4:b2:33 0800 1482: 213.107.222.172.4114 > 115.74.227.206.80: P [bad tcp cksum 7592!] 0:1428(1428) ack 1 win 17520 (frag 46920:1448@0+) (ttl 111, len 1468, bad cksum b452!)

Source of trace

This detect is from the raw logs file http://www.incidents.org/logs/Raw/2002.8.15

I can only speculate on the network layout. If you could believe the logs one would say that the IDS that captured these logs was on the 115.74.0.0 class b subnet. Since all the packets have either a source IP address or destination IP address on that subnet and the addresses cover a wide range of that subnet (from 115.74.9.61 -> 115.74.249.202). But the addresses have been obfuscated. Which explains the checksum errors in the tcpdump output. The link level header shows the packets were received from MAC address 00:03:e3:d9:26:c0 and the destination MAC address is 00:00:0c:04:b2:33. The first six digits indicate the vendor. Using http://coffer.com/mac_find/ I found that both of these addresses belong to Cisco. So we know the IDS is likely between two routers. The packets to port 80 on internal machines are from established TCP connections and there are packets in the logs from port 80 on a machine on the 115.74.0.0 network so there are web servers on the internal network.

internet --- cisco rtr ------- cisco rtr ------ web servers
|
IDS

Detect was generated by

The detect was generated by the Snort intrusion detection system. I re-ran Snort on the raw logs using the following command:

snort -d -l ./logs -c /downloads/snort-1.9.1/etc/snort.conf -r 2002.8.15

and got the following output:

[**] [1:1322:4] BAD TRAFFIC bad frag bits [**]
[Classification: Misc activity] [Priority: 3]
09/14-19:34:12.036507 213.107.222.172 -> 115.74.227.206
TCP TTL:111 TOS:0x0 ID:42741 IpLen:20 DgmLen:1468 DF MF
Frag Offset: 0x0000 Frag Size: 0x05A8

[**] [1:1322:4] BAD TRAFFIC bad frag bits [**]
[Classification: Misc activity] [Priority: 3]
09/14-19:34:17.826507 213.107.222.172 -> 115.74.227.206
TCP TTL:111 TOS:0x0 ID:43027 IpLen:20 DgmLen:1468 DF MF
Frag Offset: 0x0000 Frag Size: 0x05A8

[**] [1:1322:4] BAD TRAFFIC bad frag bits [**]
[Classification: Misc activity] [Priority: 3]
09/14-19:34:29.306507 213.107.222.172 -> 115.74.227.206
TCP TTL:111 TOS:0x0 ID:43550 IpLen:20 DgmLen:1468 DF MF
Frag Offset: 0x0000 Frag Size: 0x05A8

[**] [1:1322:4] BAD TRAFFIC bad frag bits [**]
[Classification: Misc activity] [Priority: 3]
09/14-19:34:52.586507 213.107.222.172 -> 115.74.227.206
TCP TTL:111 TOS:0x0 ID:44682 IpLen:20 DgmLen:1468 DF MF
Frag Offset: 0x0000 Frag Size: 0x05A8

[**] [1:1322:4] BAD TRAFFIC bad frag bits [**]
[Classification: Misc activity] [Priority: 3]
09/14-19:35:38.946507 213.107.222.172 -> 115.74.227.206
TCP TTL:111 TOS:0x0 ID:46920 IpLen:20 DgmLen:1468 DF MF
Frag Offset: 0x0000 Frag Size: 0x05A8

A description of Snorts Fragbits checking can be found at http://www.snort.org/docs/writing_rules/chap2.html#tth_sEc2.3.7. Snort logged these packets because both the 'Don't fragment' and the 'More fragments' bits were set. Packets with the 'don't fragment' bit set should never get fragmented. The ‘more fragments’ bit indicates that this is a packet fragment and additional fragments will follow. A router will respond with a 'ICMP_UNREACH_NEEDFRAG' ICMP message (type 3 code 4) if it receives a packet that has the don't fragment bit set and is larger than the MTU size of the interface the router would want to send the packet to.

Probability the source address was spoofed

The illegal fragment bits indicates that the packet was probably crafted but the rest of the fields in the packet seem reasonable. A TTL of 111 likely indicates the source is 17 (128-111) hops away from the IDS. The total length field in the IP header of each packet is 0x5BC or 1468 bytes. This corresponds to the 'Frag Size' of 1448 (0x5A8) which is the total length minus the IP header length. For ethernet the maximum transmission unit (MTU) for an IP datagram is 1500 bytes so this is a valid ethernet packet.

However as described below the payload of these packets contain the signature of the code red attack. There are a total of 13 fragmented code red packets in this particular log file. They all have the same illegal fragment bits and the same packet size. They also have similar numbers in the TTL field (108-112) but the source IP addresses are different. A lookup of the source IP addresses using http://www.dnsstuff.com/ yielded the following:

IP Address	Country	Description
213.107.222.172	UNITED KINGDOM	NTL ADAM solution at Winnersh
213.106.223.199	UNITED KINGDOM	NTL Internet - Brentford site
202.69.163.202	PHILIPPINES	ComClark Network & Technology Corp.
80.32.49.162	SPAIN	Provider Local Registry

DShield (WWW.dshield.org) didn't report any attacks attributed to these addresses.

Although code red requires a response from the machine being attacked the different IP addresses indicates the source address may have been spoofed.

Description of attack

Although Snort logged the packets because of an illegal combination of fragmentation bits, a look inside the packet shows that it is actually a Code Red attack. Using tcpdump as follows:

tcpdump -n -X -x -r 2002.8.15 > 2002.8

gives the output below:

```
0x0000   4500 05bc a6f5 6000 6f06 c4a5 d56b deac      E.....`.o....k..
0x0010   734a e3ce 1012 0050 4d44 b3de 5499 863e      sJ.....PMD..T..>
0x0020   5018 4470 00fb 0000 2f64 6566 6175 6c74      P.Dp..../default
0x0030   2e69 6461 3f4e 4e4e 4e4e 4e4e 4e4e 4e4e      .ida?NNNNNNNNNNN
0x0040   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x0050   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x0060   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x0070   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x0080   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x0090   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
```

```
0x00a0   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x00b0   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x00c0   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x00d0   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x00e0   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x00f0   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x0100   4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e      NNNNNNNNNNNNNNNN
0x0110   4e4e 4e4e 4e25 7539 3039 3025 7536 3835      NNNNN%u9090%u685
0x0120   3825 7563 6264 3325 7537 3830 3125 7539      8%ucbd3%u7801%u9
0x0130   3039 3025 7536 3835 3825 7563 6264 3325      090%u6858%ucbd3%
0x0140   7537 3830 3125 7539 3039 3025 7536 3835      u7801%u9090%u685
0x0150   3825 7563 6264 3325 7537 3830 3125 7539      8%ucbd3%u7801%u9
0x0160   3039 3025 7539 3039 3025 7538 3139 3025      090%u9090%u8190%
```

The characters "default.ida?NNNNNN" is a signature of Code Red.

Attack mechanism

The Code Red attack has been well documented (see http://www.ciac.org/ciac/bulletins/l-117.shtml) so I'm not going to described it in detail here. It is the fragmentation that differentiates this particular attack. There are 34 other packets in the same log file that have illegal fragmentation. Of these all but 3 have the same length and offset. Not all of the packets with illegal fragmentation have a code red payload but a high percentage do.

Some possible causes of this detect are:

IDS Evasion

Since most IDS's are signature based they look for particular patterns in the packets. If the particular attack pattern the IDS is looking for is spread over several packets (via fragmentation) then some IDS's won't detect the attack. An excellent article on IDS evasion can be found at http://secinf.net/info/ids/idspaper/idspaper.html.
The illegal fragmentation bits in this detect suggest that these are crafted packets which would lead one to suspect this is an IDS evasion attempt. There are several fragmentation tools available including one by Dug Song (http://packetstorm.widexs.nl/UNIX/IDS/nidsbench/fragrouter.html). None that I know of would set both “don’t fragment” and “more fragments” in the same packet however. It could be someone was trying to launch a code red attack using a new fragmentation tool to evade detection. If this is the case though the attacker not only made the mistake of setting the fragmentation bits improperly but also didn't fragment the attack into small enough packets to be able to evade vulnerable IDS's. If they are crafted packets the attacker was smart enough to change the IP identification number. The IP checksum also changes but of course since other fields in the packet changed the checksum would have to or the packet would have been dropped by the first router it went to on its way to the destination.
Another possible explanation for seeing the same packet with different IP ID’s is that these are TCP retransmissions. The time between the packets in the Snort output above is 5, 12, 23, and 48 seconds. Typically retransmissions occur at (3, 6, 12, 24,

48, etc.) intervals. Varying delays and the possibility that one packet was lost all together could account for the difference between the expected and actual times so it's quite possible that these were retransmissions and not crafted packets being fragmented to avoid detection.

Denial of Service

The fact that the offset for most of the fragmented packets is zero could lead one to suspect a denial of service attack. In other words they may be trying to get a server to queue up these fragments waiting on the rest of the packet until the server runs out of memory. However there are not enough packets for this to be feasible and why use a code red payload in this case. Also most servers will discard these packets after some time period anyway.

Scanning

Perhaps the sender of these packets was hoping to see an “ICMP Fragment Reassembly Time Exceeded” (Type:11 Code:1 TTL EXCEEDED) from the targeted machine. But it doesn’t make sense that they would use a code red payload for this either.

Of course it is also possible that this is not an attack or scan but the result of a flawed router that set the bits incorrectly. But with so many of the fragmented packets containing the code red payload this seems unlikely. Although there are problems with all the possibilities I’ve mentioned I believe the most likely cause of these packets is IDS evasion.

Correlations

As mentioned earlier Dshield showed no activity from any of the source IP’s.

Similar fragmentation was attributed to a DOS attack in http://lists.jammed.com/incidents/2001/07/0085.html

Donald Merchant in his practical attributes this to faulty hardware http://www.giac.org/practical/Donald_Merchant_GCIA.doc

Robert Buckley saw a similar detect that had the DF and MF bits set and the same fragment size and offset (http://archives.neohapsis.com/archives/incidents/2002-04/0054.html). This was directed at his DNS server and the server did eventually return an ICMP Fragment Reassembly Time Exceeded response. The packets he saw do not contain code red however.

Peter Szczepankiewicz submitted a similar detect (http://cert.uni-stuttgart.de/archive/intrusions/2002/11/msg00019.html) that he attributed to reconnaissance.

Roger Thompson mentions that DF MF has been seen with worms before, and it is common. But there is no final explanation of why an attacker would chose to do this.

http://cert.uni-stuttgart.de/archive/intrusions/2002/08/msg00239.html

Scott Gregory suggests that Code Red packets are fragmented to avoid IDS detection.
http://cert.uni-stuttgart.de/archive/intrusions/2002/08/msg00106.html

Evidence of active targeting

Several destinations received the fragmented packets including the following:

IP Address	Destination
213.107.222.172	115.74.227.206
213.106.223.199	115.74.71.133
202.69.163.202	115.74.158.86
80.32.49.162	115.74.164.169

In each case port 80 was targeted. If I knew these destinations were web servers I would suspect active targeting but as it is there is not enough information to be sure.

Severity

Criticality - 3 - Criticality measures how critical the targeted system is. Since many of the fragmented packets were repeated several times one could conclude that the targeted machines didn't respond and may not even exist. None of the addresses that received fragmented code red packets appear elsewhere in the logs. If these machines do exist it's not known what information is on the particular web server in question but many web servers carry sensitive information or are an important part of a companies e-commerce. With of all the unknowns I will give this detect a Criticality of 3.

Lethality - 3 - Lethality measures how severe the damage would be if the attack succeeded. This could be looked at in two ways for this particular detect. If the fragmenting had succeeded in evading the IDS or if the Code Red exploit has succeeded. The "default.ida?NNNNNN" signature in the fragmented packets indicates that the web server is being probed for the vulnerability that Code Red exploits. It does not mean that a code red attack would necessarily be successful. So if the fragmenting had been done correctly and the probe had worked then the attacker would know if the web server was vulnerable and the victim would not know that he had been scanned. If the server was vulnerable then it would allow the web server to be compromised. Viewing the attack as a IDS evasion attempt I would rate it's lethality a 3.

System Countermeasures - 3 - Only unpatched IIS servers are vulnerable to Code Red but as mentioned above the fragmented packets are the only ones to or from these address and they don't appear to have responded so we know nothing about the machines.

Network Countermeasures - 3 - In this case Snort caught the illegal fragments and even if the fragmentation had been successful it would have still caught the Code Red attack as long as the frag2 preprocessor was being used which is the default. However http://lists.jammed.com/pen-test/2002/04/0058.html discusses a bug in Snort version 1.8.6 and below that involves fragmentation. Border routers and firewalls can be used to limit access but since the source addresses were from all over the globe this university is apparently allowing access to everyone (assuming the IDS was not outside the router and firewall). Again there is not much information to go on but at least they have an IDS so I will rate Network Countermeasures a 3.

(3+3) - (3 + 3) = 0

Defensive recommendation

Make sure the latest version of Snort is being used and make sure all web servers have the latest patches. The university might also consider blocking fragmented packets depending upon whether they are “normally” received or needed. They could also block some of the ICMP responses which the attacker might have been trying to elicit.

Multiple choice test question

What is the most likely reason the following packet was logged by Snort

09/14-19:34:12.036507 213.107.222.172 -> 115.74.227.206
TCP TTL:111 TOS:0x0 ID:42741 IpLen:20 DgmLen:1468 DF MF
Frag Offset: 0x0000 Frag Size: 0x05A8

a) The Type of Service is 0
b) The fragment offset is 0
c) Illegal combination of IP flags
d) Use of a reserved IP address

Answer - c

Detect #2 RPC portmap request mountd

16:46:45.956507 66.1.161.243.600 > 32.245.170.117.111: udp 56
16:46:45.966507 66.1.161.243.600 > 32.245.170.117.111: udp 56
16:46:50.776507 66.1.161.243.600 > 32.245.170.117.111: udp 56
16:46:50.926507 66.1.161.243.600 > 32.245.170.117.111: udp 56
16:46:59.046507 66.1.161.243.600 > 32.245.170.117.111: udp 56
16:46:59.226507 66.1.161.243.600 > 32.245.170.117.111: udp 56
16:47:03.756507 66.1.161.243.600 > 32.245.170.117.111: udp 56
16:47:04.116507 66.1.161.243.600 > 32.245.170.117.111: udp 56

Source of trace

This detect is from the raw logs file http://www.incidents.org/logs/Raw/2002.9.23

Again I can only speculate on the network layout. The internal network is 32.245.0.0/24 after the addresses were changed. The link level headers again show the packets were between the same two routers as in detect #1:

```
internet --- cisco rtr ------- cisco rtr ------ 32.245.0.0/16
                    |
                   IDS
```

Detect was generated by

The detect was generated by the Snort intrusion detection system. I re-ran Snort on the raw logs using the following command:

snort -d -l ./logs -c /downloads/snort-1.9.1/etc/snort.conf -r 2002.9.23

and got the following output:

```
[**] [1:579:2] RPC portmap request mountd [**]
[Classification: Decode of an RPC Query] [Priority: 2]
10/23-17:46:45.956507 66.1.161.243:600 -> 32.245.170.117:111
UDP TTL:109 TOS:0x0 ID:41759 IpLen:20 DgmLen:84
Len: 64
[Xref => arachnids 13]

[**] [1:579:2] RPC portmap request mountd [**]
[Classification: Decode of an RPC Query] [Priority: 2]
10/23-17:46:45.966507 66.1.161.243:600 -> 32.245.170.117:111
UDP TTL:109 TOS:0x0 ID:42015 IpLen:20 DgmLen:84
Len: 64
[Xref => arachnids 13]

[**] [1:581:2] RPC portmap request pcnfsd [**]
[Classification: Decode of an RPC Query] [Priority: 2]
10/23-17:46:50.776507 66.1.161.243:600 -> 32.245.170.117:111
UDP TTL:109 TOS:0x0 ID:42527 IpLen:20 DgmLen:84
Len: 64
[Xref => arachnids 22]

[**] [1:581:2] RPC portmap request pcnfsd [**]
[Classification: Decode of an RPC Query] [Priority: 2]
10/23-17:46:50.926507 66.1.161.243:600 -> 32.245.170.117:111
UDP TTL:109 TOS:0x0 ID:42783 IpLen:20 DgmLen:84
```

Len: 64
[Xref => arachnids 22]

[**] [1:579:2] RPC portmap request mountd [**]
[Classification: Decode of an RPC Query] [Priority: 2]
10/23-17:46:59.046507 66.1.161.243:600 -> 32.245.170.117:111
UDP TTL:109 TOS:0x0 ID:44831 IpLen:20 DgmLen:84
Len: 64
[Xref => arachnids 13]

[**] [1:579:2] RPC portmap request mountd [**]
[Classification: Decode of an RPC Query] [Priority: 2]
10/23-17:46:59.226507 66.1.161.243:600 -> 32.245.170.117:111
UDP TTL:109 TOS:0x0 ID:45087 IpLen:20 DgmLen:84
Len: 64
[Xref => arachnids 13]

[**] [1:581:2] RPC portmap request pcnfsd [**]
[Classification: Decode of an RPC Query] [Priority: 2]
10/23-17:47:03.756507 66.1.161.243:600 -> 32.245.170.117:111
UDP TTL:109 TOS:0x0 ID:45343 IpLen:20 DgmLen:84
Len: 64
[Xref => arachnids 22]

[**] [1:581:2] RPC portmap request pcnfsd [**]
[Classification: Decode of an RPC Query] [Priority: 2]
10/23-17:47:04.116507 66.1.161.243:600 -> 32.245.170.117:111
UDP TTL:109 TOS:0x0 ID:45599 IpLen:20 DgmLen:84
Len: 64
[Xref => arachnids 22]

A description of Snort's RPC checking can be found at http://www.snort.org/docs/writing_rules/chap2.html#tth_sEc2.3.7. The following rules are what caused these packets to be logged:

alert tcp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap request mountd"; content:"|01 86 A5 00 00|";offset:40;depth:8; reference:arachnids,13; classtype:rpc-portmap-decode; flow:to_server,established; sid:1266; rev:4;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap request pcnfsd"; content:"|02 49 f1 00 00|";offset:40;depth:8; reference:arachnids,22; classtype:rpc-portmap-decode; flow:to_server,established; sid:1268; rev:4;)

Mountd uses RPC Program Number 100005 (0x0186a5) hence the rule is checking for content of "01 86 A5 00 00". And pcnfsd uses RPC Program Number 150001 (0x0249f1) hence the rule is checking for content of "02 49 f1 00 00". The two zero bytes after the program number are the first two bytes of the four byte version number. This helps reduce false alarms since the latest version numbers for mountd and pcnfsd are in the single digits.

Probability the source address was spoofed

Using http://remote.12dt.com/rns/ I found the source address (66.1.161.243) belongs to Sprint Broadband Direct. Nothing about the packets make them look crafted. The TTL (109) is reasonable. The IP ID field is incrementing, etc. Based on this and the fact that most RPC related attacks require the attacker to get responses I don't believe the address was spoofed.

Description of attack

There are numerous attacks related to RPC and to mountd and pcnfsd in particular. See "http://www.cert.org/advisories/CA-1996-08.html" and "http://www.cert.org/advisories/CA-1998-12.html" for examples. Most of these attacks exploit buffer overflows.

The source port is 600 and according to http://www.shmoo.com/mail/firewalls/mar00/msg00020.shtml" port 600 is the default port for pcserver. And according to "http://www.doc.ic.ac.uk/~mac/manuals/hpux-manual-pages/hpux/usr/man/man1m/pcserver.1m.html" pcserver is the hostside server program for Basic Serial and AdvanceLink, and advanceLink is a terminal emulation program that also supports file transfers between a PC and host system over various physical connections.

But according to "http://www.first.org/events/progconf/2000/D1-03.pdf" port 600 is also a favorite port for script kiddies. Therefore this detect could be the output of a script searching for machines running mountd or pcnfsd.

There is also a report at http://lists.jammed.com/incidents/2001/05/0100.html where Brad Doctor and Martin Markgraf describes a worm that uses port 600 to do RPC scans in an attempt to find vulnerabilities in order to spread.

With Snort rules that check the payload for a particular string it's always possible to get false alarms if the pattern occurs naturally. In this case though the rest of the packet looks like the RPC request packets I saw while testing the code I wrote for Part I of this report. For example the '02' in byte 0x27 below is the RPC version number which is typically 2.

```
16:46:45.966507 66.1.161.243.600 > 32.245.170.117.111: udp 56
0x0000   4500 0054 a41f 0000 6d11 e502 4201 a1f3        E..T....m...B...
0x0010   20f5 aa75 0258 006f 0040 a599 ff05 862f        ...u.X.o.@...../
0x0020   0000 0000 0000 0002 0001 86a0 0000 0002        ................
0x0030   0000 0003 0000 0000 0000 0000 0000 0000        ................
0x0040   0000 0000 0001 86a5 0000 0001 0000 0011        ................
0x0050   0000 0000                                      ....
16:46:50.776507 66.1.161.243.600 > 32.245.170.117.111: udp 56
0x0000   4500 0054 a61f 0000 6d11 e302 4201 a1f3        E..T....m...B...
0x0010   20f5 aa75 0258 006f 0040 e24a ff05 8630        ...u.X.o.@.J...0
0x0020   0000 0000 0000 0002 0001 86a0 0000 0002        ................
0x0030   0000 0003 0000 0000 0000 0000 0000 0000        ................
```

0x0040 0000 0000 0002 49f1 0000 0002 0000 0011 .....I.....
0x0050 0000 0000

It's also possible that someone has a misconfigured system and is targeting the university's network unintentionally but there was no other RPC traffic in the logs and with a source port of 600 it sounds suspiciously like the worm described by Brad Doctor and Martin Markgraf.

Attack mechanism

As mentioned earlier the RPC portmapper, which listens on port 111, is used to find the RPC services running on a server. The packets in this detect are querying the mountd and pcnfs services as seen by the 0186a5 and 0249f1 in the payloads.

If the services were found the attacker could then attempt to exploit the buffer overflow vulnerabilities in mountd and pcnfsd which could then give the attacker root access to the machine. The logs don't show a response from the portmapper but since the response doesn't include the program number it would not have been logged by Snort anyway so we don't know if this machine was running the queried services or not. There are no other packets in the logs involving 66.1.161.243 but again we don't know if that's because there were none or if they just didn't match any of the Snort rules.

Correlations

A search of the source address using Dshield didn't yield anything, but mynetwatchman had two incidents that were closed. I assume this means that they were either false or were appropriately investigated and dealt with.

In http://cert.uni-stuttgart.de/archive/intrusions/2003/01/msg00209.html Mark Donaldson attributes this attack to RPC reconnaissance.

In http://www.giac.org/practical/GCIA/Doug_Kite_GCIA.pdf Doug Kite similarly attributes this attack to someone looking for a machine running mountd or pcnfsd that can be exploited.

I posted my analysis to incidents.org on 5-30-03 but didn't receive any responses.

Evidence of active targeting

There were no other packets in the log from the same source address so it doesn't look like anyone at that address was targeting the university for attack.
Also the worm that I've attributed this to uses a random number generator to select its victims so I don't believe there was active targeting in this case.

Severity

Criticality - 3 - Criticality measures how critical the targeted system is. Since this worm targets random machines it's unknown how critical the machine it chooses will be, so I will give this detect a Criticality of 3.

Lethality - 3 - Lethality measures how severe the damage would be if the attack succeeded. As mentioned earlier there are attacks against mountd that can lead to the attacker getting root access to the machine. But in this case it was just a probe and not an actual attack so I will give this detect a Lethality of 3.

System Countermeasures - 3 - The only other packets to 32.245.170.117 are scans. There are no packets logged from 32.245.170.117 so we know little about it. At least we don't have any evidence that it is horribly configured (i.e. responding to everything).

Network Countermeasures - 3 - We know there is a border router in this network but we don't know how well configured it is. The fact that packets to so many different ports are getting through leads me to believe it probably doesn't have a very robust access control list. But since they do have an IDS on the network they are security aware so I will give a 3 for Network Countermeasures.

(3+3) - (3+3) = 0

Defensive recommendation

Make sure all RPC daemons have the latest patches. Disable any services that aren't needed. If RPC related services are only needed by known IP addresses then block access to the portmapper from other addresses with a border router or firewall. Also Wietse Venema at http://ftp.porcupine.org/pub/security/ proposes to have a secure version of the portmapper that should be looked into if RPC is needed.

Multiple choice test question

Which of the following ports is commonly used for RPC portmap requests:

a) 111
b) 800
c) 1010
d) 100

Correct answer is (a).

Detect #3 XMAS scan

Source of trace

This detect is one I captured on a network that looks as follows:

```
internet ------ cisco rtr ------ firewall ------ web/ftp servers
                      |
                     IDS
```

Detect was generated by

The IDS being used was Snort version 1.9.1.

I used the default Snort rules and the following to capture the alerts:

snort -b -l logs -c /etc/snort.conf

For this detect I decided to analyse a scan. At first I used the default Snort rules and saw many scans like the following:

[**] [117:1:1] (spp_portscan2) Portscan detected from 216.8.128.71: 21 targets 21 ports in 3 seconds [**]
04/16-18:11:29.448590 216.8.128.71:1064 -> xxx.xxx.xxx.224:137
UDP TTL:115 TOS:0x0 ID:52964 IpLen:20 DgmLen:78
Len: 58

Eventually I saw something a little more interesting. A SYN/FIN scan from Chile:

[**] [111:13:1] (spp_stream4) STEALTH ACTIVITY (SYN FIN scan) detection [**]
04/25-00:56:26.157150 146.83.34.8:21 -> xxx.xxx.xxx.2:21
TCP TTL:19 TOS:0x0 ID:39426 IpLen:20 DgmLen:40
******SF Seq: 0x59E915BA Ack: 0x71F7C84F Win: 0x404 TcpLen: 20

[**] [111:13:1] (spp_stream4) STEALTH ACTIVITY (SYN FIN scan) detection [**]
04/25-00:56:26.176939 146.83.34.8:21 -> xxx.xxx.xxx.3:21
TCP TTL:19 TOS:0x0 ID:39426 IpLen:20 DgmLen:40
******SF Seq: 0x59E915BA Ack: 0x71F7C84F Win: 0x404 TcpLen: 20

[**] [111:13:1] (spp_stream4) STEALTH ACTIVITY (SYN FIN scan) detection [**]
04/25-00:56:26.237112 146.83.34.8:21 -> xxx.xxx.xxx.6:21
TCP TTL:19 TOS:0x0 ID:39426 IpLen:20 DgmLen:40
******SF Seq: 0x59E915BA Ack: 0x71F7C84F Win: 0x404 TcpLen: 20

...

[**] [111:13:1] (spp_stream4) STEALTH ACTIVITY (SYN FIN scan) detection [**]
04/25-00:56:31.196417 146.83.34.8:21 -> xxx.xxx.xxx.254:21
TCP TTL:19 TOS:0x0 ID:39426 IpLen:20 DgmLen:40
******SF Seq: 0x3EA43329 Ack: 0x79BDA771 Win: 0x404 TcpLen: 20

[**] [111:13:1] (spp_stream4) STEALTH ACTIVITY (SYN FIN scan) detection [**]
04/25-00:56:31.217067 146.83.34.8:21 -> xxx.xxx.xxx.255:21
TCP TTL:19 TOS:0x0 ID:39426 IpLen:20 DgmLen:40
******SF Seq: 0x3EA43329 Ack: 0x79BDA771 Win: 0x404 TcpLen: 20

Using a reverse DNS lookup 146.83.34.8 resolved to c1.sociales.uchile.cl.

This was obviously someone looking for an ftp server to exploit. The fact that a few destination addresses were skipped makes me think Snort is probably dropping packets occasionally.

After a couple more days I saw a scan that wasn't so obvious. It was an XMAS scan of an odd collection of ports. It is this detect that I chose to analyse.

[**] [111:10:1] (spp_stream4) STEALTH ACTIVITY (XMAS scan) detection[**]
05/22-21:27:09.457037 218.18.36.105:63207 -> xxx.xxx.xxx.6:12543
TCP TTL:42 TOS:0x0 ID:4369 IpLen:20 DgmLen:60
U*PF Seq: 0x12121212 Ack: 0x0 Win: 0x0 TcpLen: 40 UrgPtr: 0x0
TCP Options (4) => WS: 10 MSS: 265 TS: 1061109567 0 EOL

Probability the source address was spoofed

Using http://www.dshield.org/ipinfo.php I found that the source address (218.18.36.105) was registered to a Chinese telecom.

inetnum: 218.13.0.0 - 218.18.255.255
netname: CHINANET-GD
country: CN
descr: CHINANET Guangdong province network
Data Communication Division
China Telecom
admin_c: CH93-AP
tech_c: WM12-AP
remarks:
mnt_by: MAINT-CHINANET
changed: hostmaster@ns.chinanet.cn.net 20010528
status: ALLOCATED PORTABLE
source: APNIC

notify:
mnt_lower: MAINT-CHINANET-GD
rev_srv:
start: 3658285056
end: 3658678271
diff: 393215
person: WU MIAN
address:
NO.1,RO.DONGYUANHENG,YUEXIUNAN,GUANGZHOU
country: CN
phone: +086-20-83877223
fax_no: +86-20-83877223
e_mail: ipadm@gddc.com.cn
nic_hdl: WM12-AP
mnt_by: MAINT-CHINANET-GD
changed: ipadm@gddc.com.cn 20010820
source: APNIC
remarks:
notify:

I also checked http://www.mynetwatchman.com/ and this source address had two events associated with it, the most recent on May 20, 2003. These were port 80 scanning events that were attributed to “probable Nimda/Code Red”.

In the scan I saw the sequence number was always 0x12121212 and the IP ID was always 4369 so the packets were obviously crafted, but a scan requires the sender to see the response so I doubt the source address was spoofed.

Description of attack

[**] [111:10:1] (spp_stream4) STEALTH ACTIVITY (XMAS scan) detection[**]
05/22-21:27:09.457037 218.18.36.105:63207 -> xxx.xxx.xxx.6:12543
TCP TTL:42 TOS:0x0 ID:4369 IpLen:20 DgmLen:60
U*PF Seq: 0x12121212 Ack: 0x0 Win: 0x0 TcpLen: 40 UrgPtr: 0x0
TCP Options (4) => WS: 10 MSS: 265 TS: 1061109567 0 EOL

[**] [111:10:1] (spp_stream4) STEALTH ACTIVITY (XMAS scan) detection[**]
05/22-21:27:09.458536 218.18.36.105:63207 -> xxx.xxx.xxx.6:12543
TCP TTL:42 TOS:0x0 ID:4369 IpLen:20 DgmLen:60
U*PF Seq: 0x12121212 Ack: 0x0 Win: 0x0 TcpLen: 40 UrgPtr: 0x0
TCP Options (4) => WS: 10 MSS: 265 TS: 1061109567 0 EOL

[**] [111:10:1] (spp_stream4) STEALTH ACTIVITY (XMAS scan) detection[**]
05/22-21:27:09.461141 218.18.36.105:63207 -> xxx.xxx.xxx.6:12543
TCP TTL:42 TOS:0x0 ID:4369 IpLen:20 DgmLen:60
U*PF Seq: 0x12121212 Ack: 0x0 Win: 0x0 TcpLen: 40 UrgPtr: 0x0
TCP Options (4) => WS: 10 MSS: 265 TS: 1061109567 0 EOL

[**] [111:10:1] (spp_stream4) STEALTH ACTIVITY (XMAS scan) detection[**]

05/22-21:27:34.685568 218.18.36.105:64993 -> xxx.xxx.xxx.14:12543
TCP TTL:42 TOS:0x0 ID:4369 IpLen:20 DgmLen:60
U*PF Seq: 0x12121212 Ack: 0x0 Win: 0x0 TcpLen: 40 UrgPtr: 0x0
TCP Options (4) => WS: 10 MSS: 265 TS: 1061109567 0 EOL

[**] [111:10:1] (spp_stream4) STEALTH ACTIVITY (XMAS scan) detection[**]
05/22-21:27:34.689525 218.18.36.105:64993 -> xxx.xxx.xxx.14:12543
TCP TTL:42 TOS:0x0 ID:4369 IpLen:20 DgmLen:60
U*PF Seq: 0x12121212 Ack: 0x0 Win: 0x0 TcpLen: 40 UrgPtr: 0x0
TCP Options (4) => WS: 10 MSS: 265 TS: 1061109567 0 EOL

It continues on, scanning several more addresses. It mostly scans for port 12543 but also occasionally scans ports 50, 51, 52, 53, 110, 112, 3389, and 5631. Most of the scans are the XMAS variety but some are NULL scans.

The above mentioned ports were scanned the following number of times:
Port 50 - 1
Port 51 - 10
Port 52 - 20
Port 53 - 12
Port 110 - 3
Port 112 - 3
Port 3389 - 4
Port 5631 - 37
Port 12543 – 196

The following addresses were scanned
xxx.xxx.xxx.6 -> xxx.xxx.xxx.125

Attack mechanism

There is a very good article on the various scans at http://www.computercops.biz/modules.php?name=nmap.

An excerpt from this on the XMAS scan is shown below:

> "The idea is that closed ports are required to reply to your probe packet with an RST, while open ports must ignore the packets in question (see RFC 793 pp 64). The FIN scan uses a bare (surprise) FIN packet as the probe, while the Xmas tree scan turns on the FIN, URG, and PUSH flags. The Null scan turns off all flags. "

The XMAS scan is basically an attempt by the scanners to stay ahead of the IDS's. At first IDS's detected scans by looking for complete TCP connections so the scanners used the half-open scan by just sending the SYN packet. When IDS's started looking for packets with just the SYN flag set the scanners went to the SYN/FIN scan. The XMAS scan is just the next attempt at evasion.

Also noted in the article mentioned above is that this attack does not work against several operating systems including Microsoft Windows, Cisco, BSDI, HP/UX, MVS, and IRIX because these operating system do not follow RFC 793 correctly.

The thing that makes this scan different from the rest that I saw are the ports it was scanning. As defined by IANA these ports were meant to be used for the following:

50/tcp Remote Mail Checking Protocol
51/tcp IMP Logical Address Maintenance
52/tcp XNS Time Protocol
53/tcp Domain Name Server
110/tcp Post Office Protocol - Version 3
112/tcp McIDAS Data Transmission Protocol
3389/tcp MS WBT Server
5631/tcp pcANYWHEREdata
12543/tcp undefined

There are many known attacks against POP, DNS, MS WBT Server, and pcANYWHEREdata but I couldn't find any on the other ports.

However from the following web sites on trojans I found other known uses for some of these ports.
http://www.simovits.com/sve/nyhetsarkiv/1999/nyheter9902.html
http://www.tigertools.net/trojans.txt

Port 50 DRAT
Port 52 MuSka52, Skun
Port 53 ADM worm, li0n, MscanWorm, MuSka52
Port 110 ProMail trojan, ADM worm

These trojans are described at the following sites:
DRAT - http://www.saintcorporation.com/demo/saint_tutorials/backdoor_found.html
MuSka52 - http://www.sophos.com/virusinfo/analyses/trojmuska5213.html
Skun - http://securityresponse.symantec.com/avcenter/venc/data/backdoor.skun.html
ADM worm - http://www.redhat.com/archives/linux-security/1999-March/msg00004.html
li0n - http://ciac.llnl.gov/ciac/bulletins/l-064.shtml
MscanWorm - http://www.simovits.com/trojans/tr_data/y2169.html
ProMail trojan - http://www.psc.ru/sergey/bgtraq/STRANGEVIRUS/TROYAN/promail.htm

Of these DRAT, MuSka52, and Skun are backdoors that someone might be scanning for. The others are worms that use the indicated ports but would not have the XMAS scan signature when doing so.

Drat is a backdoor that listens for connections on port 48 and uses port 50 for file transfers. Since port 48 wasn't scanned I doubt DRAT is what they were looking for.

MuSka52 and Skun are backdoors that only run on Windows systems and as mentioned above the XMAS scan doesn't work on Windows systems.

Port 12543 was the port scanned most frequently and I found no information on backdoors or other vulnerabilities associated with this port. A check of DShield (http://www.dshield.org/port_report.php?port=12543) for instance showed no vulnerabilities and no more reports related to it then other random ports.

After the above analysis it doesn't look like there is any reason to be scanning the particular ports chosen. I believe they were just chosen at random by someone with a new hacker tool to play with. Since none of the ports scanned were open on the targeted network and the machines on the targeted network were Solaris machines the attacker would have received a RST in response to his scans to active machines and no response when the machine didn't exist except that the Cisco router and the Gauntlet firewall were configured to drop packets to all but a few select ports. Therefore the attacker did not receive a response to any of the scans.

Correlations

I posted this detect to incidents.org on 5/25/03. The feedback will be included at the end of this detect.

Evidence of active targeting

About half of the addresses on the subnet being monitored were scanned, possibly more since as mentioned earlier Snort may have been dropping packets, so I don't see any evidence of active targeting.

Severity

Criticality - 5 - Criticality measures how critical the targeted system is. The targeted system contains company private information and is used by important customers so I will give this detect a Criticality of 5.

Lethality - 1 - Lethality measures how severe the damage would be if the attack succeeded. Since the scan did not succeed it's unknown what attacks would have been used if the scan has succeeded. Since the scan itself was not lethal I will give this attack a Lethality of 1.

System Countermeasures - 5 - The machines in this system have been well patched and hardened using yassp so I will give a 5 for System Countermeasures.

Network Countermeasures - 5 - This network is protected by a border router and a firewall. The border router denies all protocols and source networks that are not required and the firewall does the same. The firewall also uses proxies for the protocols being used by this system (HTTP, FTP). None of the packets being

scanned for are permitted past the firewall so I will give a 5 for Network Countermeasures.

(5+1) - (5+5) = -4

Defensive recommendation

To protect against this type of scan one needs to make sure there is adequate network filtering in place to drop packets that are not needed by the system such that responses are not sent. For the system in question I feel strongly that this is the case.

Multiple choice test question

Which TCP flags does the XMAS scan set?

a) SYN
b) SYN, FIN
c) SYN, PUSH, FIN
d) URG, PUSH, FIN

Correct answer is (d).

Feedback from posting to incidents.org

Note: The analysis above was updated to incorporate the comments I received from the posting.

One comment was regarding the source address 218.18.36.105. Originally I had used a reverse DNS lookup utility that showed the address as unregistered. Both Andrew Jones and Oliver Viitamaki pointed me to other utilities (e.g. APINIC) that were able to find the address.

A second comment that both Andrew and Oliver made was that the scan could be used to locate active machines even if the ports being scanned were not open. That prompted me to add the information detailing how the router and firewall were configured such that they would drop the packets in this scan and not return a RST.

A third comment was regarding port 12543. Andrew Jones pointed me to information on the recent use of port 12543 at
http://www.dshield.org/port_report.php?port=12543

Assignment 3 - "Analyse This"

Summary

In this report logs from 4/17/03 – 4/21/03 were analysed. In these logs there were a total of 260908 alerts, 430685 scans, and 9673 out of spec events. The report details the top ten alerts, the most prevalent source addresses causing the alerts, the top scanners, as well as most frequent ports scanned. Many of the events appear to be false positives and therefore the rules should be tightened. Also if the IDS that generated the data is inside a border router and firewall then these devices should be more restrictive. If the IDS is outside the perimeter then moving it inside should be considered to reduce the enormous amount of data being generated. This will not only make the actual intrusions stand out it will also reduce the load on the IDS and help prevent dropped packets.

There were also many actual attacks as detailed in this report. MY.NET.201.58 is very likely infected with the Adore worm. MY.NET.235.110 is possibly infected with the Adore worm and it was also detected sending tiny fragments. The following machines were detected attempting to IRC: MY.NET.198.221, MY.NET.88.163, MY.NET.83.173, MY.NET.253.42, MY.NET.105.48.

Also the out-of-spec detection needs to be updated to take into account the fact that formerly reserved TCP flags are now being used for Explicit Congestion Notification (ECN).

Files Analysed

Alerts	Scans	Out of Spec
alert030417.gz	scans.030417.gz	oos_Report_2003_04_17_9696
alert030418.gz	scans.030418.gz	oos_Report_2003_04_18_5113
alert030419.gz	scans.030419.gz	oos_Report_2003_04_19_8227
alert030420.gz	scans.030420.gz	oos_Report_2003_04_20_16512
alert030421.gz	scans.030421.gz	oos_Report_2003_04_21_32071

List of Detects

There were a total of 260908 detects in the above alert files. The following table lists the top 10 alerts. The top 10 alerts accounted for 247043 alerts, 95% of the total.

Alert	Count	% of Total
SMB Name Wildcard	130777	50
Watchlist 000220 IL-ISDNNET-990517	42056	16
High port 65535 tcp - possible Red Worm - traffic	14917	6
CS WEBSERVER - external web traffic	13626	5
High port 65535 udp - possible Red Worm - traffic	11670	4

spp_http_decode: IIS Unicode attack detected	10920	4
UMBC NIDS IRC Alert] XDCC client detected attempting to IRC	7881	3
TFTP - Internal TCP connection to external tftp server	7832	3
EXPLOIT x86 NOOP	3797	1
spp_http_decode: CGI Null Byte attack detected	3567	1

Analysis of Top Ten Detects

SMB Name Wildcard

130653 detects were found.

Sources associated with this detect

There were a total of 22547 unique source addresses associated with this detect. All of them were external addresses.

The table below lists the top five sources.

Source	Count
194.148.17.27	2713
141.154.195.105	1482
213.135.141.39	1143
129.132.180.77	466
195.157.194.69	381

Destinations associated with this detect

There were a total of 38344 unique destination addresses associated with this detect, all of them internal addresses.

The table below lists the top five destinations.

Destination	Count
MY.NET.24.34	1492
MY.NET.12.2	1010
MY.NET.194.13	973
MY.NET.113.4	509
MY.NET.29.3	481

Description of detect

This detect is not part of the standard Snort rule set. An excellent article by Bryce Alexander (http://www.sans.org/resources/idfaq/port_137.php) describes the SMB Name Wildcard rule that is used to detect netbios name table lookups. Name table lookups are common on a local network where file sharing is being done but they would not normally originate from an external network. In these logs they all originate from an external source:

Source	Country	
194.148.17.27	SWITZERLAND	Federation International de Volleyball
141.154.195.105	US	Verizon Internet Services
213.135.141.39	RUSSIAN FEDERATION	Dynamic IP pool for access server
129.132.180.77	Switzerland	Swiss Federal Institute of Technology Zurich
195.157.194.69	UNITED KINGDOM	Takenaka Belgium N.V

If this traffic is legitimate then the Snort rules should be adjusted. Most likely NBTSTAT scanning and/or the network.vbs worm was very active during this time. But if the university is using the Netbios SMB service across its perimeter then it should adjust the Snort rules to prevent excessive false positives.

Defensive Recommendations

The NetBios ports should be blocked by a firewall or border router since NetBios is normally only needed across the local network.

Correlation

Les Gordon suggested adjusting the rule to not trigger on internal traffic.
Donald Merchant suggested using a VPN if netbios is needed into or out of the university network.

Watchlist 000220 IL-ISDNNET-990517

There were 42011 detects of this type.

Sources associated with this detect

There were a total of 125 unique external source addresses associated with this detect, all of them external.

The table below lists the top five sources.

Source	Count
212.179.116.236	14009
212.179.116.153	13401

212.179.27.6	1856
212.179.5.161	1607
212.179.105.111	1112

All of the source addresses were from the 212.179.0.0 subnet.

Destinations associated with this detect

There were a total of 187 unique destination addresses associated with this detect, all of them internal destinations.

The table below lists the top five destinations.

Destination	Count
MY.NET.209.158	14003
MY.NET.233.242	13398
MY.NET.196.161	1995
MY.NET.242.250	1191
MY.NET.84.196	1112

Description of detect

An IP lookup using http://www.dnsstuff.com/ found the following:

Country: ISRAEL (high)

ARIN says that this IP belongs to RIPE; I'm looking it up there.

Using cached answer (or, you can get fresh results).

% This is the RIPE Whois server.
% The objects are in RPSL format.
%
% Rights restricted by copyright.
% See http://www.ripe.net/ripencc/pub-services/db/copyright.html

inetnum: 212.179.0.0 - 212.179.0.255
netname: REDBACK-EQUIPMENT
mnt-by: INET-MGR
descr: BEZEQINT-EQUIPMENT
country: IL
admin-c: MR916-RIPE
tech-c: ZV140-RIPE
status: ASSIGNED PA
remarks: please send ABUSE complains to abuse@bezeqint.net
remarks: INFRA-AW
notify: hostmaster@bezeqint.net
changed: hostmaster@bezeqint.net 20021020
source: RIPE

route: 212.179.0.0/18
descr: ISDN Net Ltd.
origin: AS8551
notify: hostmaster@bezeqint.net
mnt-by: AS8551-MNT
changed: hostmaster@bezeqint.net 20020618
source: RIPE

person: Miri Roaky
address: bezeq-international
address: 40 hashacham
address: petach tikva 49170 Israel
phone: +972 1 800800110
fax-no: +972 3 9203033
e-mail: hostmaster@bezeqint.net
mnt-by: AS8551-MNT
nic-hdl: MR916-RIPE
changed: hostmaster@bezeqint.net 20021027
changed: hostmaster@bezeqint.net 20030204
source: RIPE

person: Zehavit Vigder
address: bezeq-international
address: 40 hashacham
address: petach tikva 49170 Israel
phone: +972 1 800800110
fax-no: +972 3 9203033
e-mail: hostmaster@bezeqint.net
mnt-by: AS8551-MNT
nic-hdl: ZV140-RIPE
changed: hostmaster@bezeqint.net 20021027
changed: hostmaster@bezeqint.net 20030204
source: RIPE

This rule is apparently watching for the 212.179.0.0 network. An Israel site the university must deem suspect.

In the logs I examined the ports targeted were a diverse set with no apparent pattern.

Defensive Recommendations

If this site is known for illicit activity then it should be blocked at the firewall or border router.

Correlation

Donald Gregory saw similar alerts. In his case the watchlist addresses were for file sharing sites however.

High port 65535 tcp - possible Red Worm - traffic

There were 14904 detects of this type.

Sources associated with this detect

There were 91 unique external source addresses and 64 unique internal source addresses for a total of 155 source addresses.

The table below lists the top five sources.

Source	Count
24.26.201.174	1694
MY.NET.207.230	1267
80.181.195.253	1145
64.218.242.59	1070
MY.NET.234.150	973

Destinations associated with this detect

There were 99 unique external destination addresses and 61 unique internal destination addresses for a total of 160 destination addresses.

The table below lists the top five destinations.

Destination	Count
MY.NET.228.130	1693
80.181.195.253	1267
MY.NET.207.230	1145
MY.NET.225.226	1070
MY.NET.206.130	904

Description of detect

A description of the Red worm / Adore worm can be found at http://www.f-secure.com/v-descs/adore.shtml. It is not the same as 'Code Red'. It replaces system files with Trojans and when activated listens on port 65535.

7190 times the 65535 port was an external destination port.
7597 times the 65535 port was an external source port.
77 times the 65535 port was an internal destination port.
40 times the 65535 port was an internal source port.

Since clients are usually assigned random ephemeral ports, occasionally port 65535 will be chosen so some of these are likely false alarms. But the high number of occurrences on some of the external IP's indicates these machines are probably

infected by the Adore worm. Since return traffic could cause machines being probed to appear on the top five sources list not all machines on this list are necessarily infected.

Defensive Recommendations

I wouldn't recommend blocking all 65535 traffic since that could stop some legitimate traffic. Instead I would make sure all the university's machines have all the necessary patches to protect against the Adore worm.

Correlation

Donald Cunningham in his practical also mentions the Adore worm and references http://www.nipc.gov/cybernotes/2001/cyberissue2001-07.pdf

CS WEBSERVER - external web traffic

There were 13617 detects of this type.

Sources associated with this detect

There were 4440 unique source addresses, all of them external.

The table below lists the top five sources.

Source	Count
66.77.73.236	776
218.19.126.17	205
64.68.84.147	84
66.77.73.237	82
64.68.84.49	81

Destinations associated with this detect

There was only 1 unique internal destination address and no unique external destination addresses. The internal destination being targeted was MY.NET.100.165 and in each case it was port 80 that was being targeted.

Description of detect

Apparently the university is logging all external connections to a particular internal web server.

Defensive Recommendations

Without knowing more about the universities policies it's impossible to make a definitive recommendation. If the university doesn't want external use of this machine they should block the traffic at the firewall or border router.

Correlation

I was unable to find any other practicals that analysed this detect.

High port 65535 udp - possible Red Worm - traffic

There were 11663 detects of this type.

Sources associated with this detect

There were 206 unique external source addresses and 66 unique internal source addresses for a total of 272 source addresses.

The table below lists the top five sources.

Source	Count
MY.NET.201.58	4289
66.42.68.210	2787
66.139.79.202	453
4.46.32.83	418
MY.NET.207.230	228

Destinations associated with this detect

There were 202 unique external destination addresses and 103 unique internal destination addresses for a total of 305 destination addresses.

The table below lists the top five destinations.

Destination	Count
MY.NET.201.58	4704
66.42.68.210	2474
66.139.79.202	444
4.46.32.83	387
MY.NET.207.230	232

Description of detect

As mentioned above a description of the Red worm / Adore worm can be found at http://www.f-secure.com/v-descs/adore.shtml.

1234 times the 65535 port was an external destination port.

1418 times the 65535 port was an external source port.
4813 times the 65535 port was an internal destination port.
4300 times the 65535 port was an internal source port.

These detects are more disturbing then the "High port 65535 udp - possible Red Worm - traffic" alerts because in this case it looks like internal machines are infected given the large number of cases where an internal machine is using port 65535.

Defensive Recommendations

8996 of the detects are associated with MY.NET.201.58:65535. This machine is clearly infected and should be taken offline and disinfected as soon as possible. Once this is done the university should make sure all their machines have the latest patches.

Correlation

Donald Cunningham in his practical also mentions the Adore worm and references http://www.nipc.gov/cybernotes/2001/cyberissue2001-07.pdf

spp_http_decode: IIS Unicode attack detected

There were 10913 detects of this type.

Sources associated with this detect

There were 269 unique external source addresses and 515 unique internal source addresses for a total of 784 source addresses.

The table below lists the top five sources.

Source	Count
MY.NET.197.42	421
MY.NET.84.218	371
MY.NET.88.239	327
MY.NET.106.96	324
MY.NET.97.71	191

Destinations associated with this detect

There were 654 unique external destination addresses and 194 unique internal destination addresses for a total of 848 destination addresses.

The table below lists the top five destinations.

Destination	Count
61.129.67.77	371

210.219.197.11	369
207.200.86.66	341
211.115.215.50	340
64.12.54.24	252

Description of detect

A web server unicode attack relies on substituting Unicode characters for the acsii characters that make up a URL. Frequently it's used in a directory transversal attack where it substitutes Unicode characters for the / symbol to circumvent the web servers attempt at preventing access to private directories.

Defensive Recommendations

This problem has been fixed in the latest version of all the popular web servers so insuring all the universities web servers have the latest patches will protect against this attack. What's disturbing though is that internal machines appear to be launching the attack. Closer examination of the logs shows that the internal machines are not web servers and the external machines are (based on port numbers involved). Either these are false alarms, the internal machines are infected with a virus, or someone at the university is attempting to attack external web servers.

Correlation

Donald Gregory in his practical suggests turning off the ISAPI service if it's not needed as well as installing the latest patches.

[UMBC NIDS IRC Alert] XDCC client detected attempting to IRC

There were 7872 detects of this type.

Sources associated with this detect

There were 8 unique source addresses, all of them internal.

The table below lists the top five sources.

Source	Count
MY.NET.198.221	7729
MY.NET.88.163	67
MY.NET.83.173	62
MY.NET.253.42	5
MY.NET.105.48	4

Destinations associated with this detect

There were 12 unique destination addresses, all of them external.

The table below lists the top five destinations.

Destination	Count
205.188.149.12	7729
66.202.41.240	67
157.156.254.111	62
206.62.14.7	3
66.252.2.60	2

The destination port was always 6667.

Description of detect

This alert is associated with Internet Relay Chat (IRC) traffic. IRC typically uses ports 6666, 6667, or 7000. It is often used to share music and graphics that can use up considerable network bandwidth.

Defensive Recommendations

This depends on the universities policy. Some use of IRC may be acceptable. If not the chat sites being accessed can be blocked at the firewall or border router. If only limited use is permitted then the user at MY.NET.198.221 needs to be reminded of the policy.

Correlation

Sebastien Pratte in his practical also suggested blocking IRC at the firewall.

TFTP - Internal TCP connection to external tftp server

There were 7822 detects of this type.

Sources associated with this detect

There were 31 unique external source addresses and 10 unique internal source addresses for a total of 41 source addresses.

The table below lists the top five sources.

Source	Count
MY.NET.239.214	1071
MY.NET.251.70	1024
MY.NET.201.42	1017
64.12.30.224	1014
MY.NET.223.114	611

Destinations associated with this detect

There were 33 unique external destination addresses and 10 unique internal destination addresses for a total of 43 destination addresses.

The table below lists the top five destinations.

Destination	Count
MY.NET.239.214	1096
MY.NET.201.42	1080
81.5.166.85	1024
64.12.30.224	1000
MY.NET.223.114	759

Description of detect

As the alert describes this is detecting internal TCP connections to external tftp servers. Trivial file transfer protocol (tftp) uses UDP to transfer files. Again this could be harmless or it could be abused. Also some viruses/worms such as Nimda use tftp to spread.

Defensive Recommendations

This again depends on the universities policy. If tftp is prohibited then it should be blocked at the firewall or border router. If not then this rule should probably be replaced by a more specific rule for detecting Nimda to avoid so many false positives.

Correlation

Aaron Hackworth in his practical also suggested blocking tftp traffic unless there is a strong need for it.

EXPLOIT x86 NOOP

There were 3794 detects of this type.

Sources associated with this detect

There were 123 unique source addresses, all of them external.

The table below lists the top five sources.

Source	Count
217.224.228.225	949
217.224.228.7	910
68.70.85.172	592

194.204.114.25	425
217.235.25.131	182

Destinations associated with this detect

There were 160 unique destination addresses, all of them internal.

The table below lists the top five destinations.

Destination	Count
MY.NET.86.19	1444
MY.NET.198.227	428
MY.NET.5.44	269
MY.NET.5.45	227
MY.NET.5.67	182

Description of detect

This detect looks for X86 NOOP instructions in the packet payload. This can indicate a buffer overflow attempt (see http://www.cccure.org/amazon/idssignature.pdf).

Defensive Recommendations

Make sure all machines have the latest patches to protect against buffer overflow vulnerabilities. Also ensure perimeter devices filter unnecessary packets and internal machines only run needed services.

Correlation

Aaron Hackworth suggested ensuring the Snort rule is up to date – Snort ID 684

spp_http_decode: CGI Null Byte attack detected

There were 3563 detects of this type.

Sources associated with this detect

There were 27 unique external source addresses and 103 unique internal source addresses for a total of 130 source addresses.

The table below lists the top five sources.

Source	Count
MY.NET.195.155	343
MY.NET.236.254	342
MY.NET.238.2	288

MY.NET.194.125	278
MY.NET.235.34	229

Destinations associated with this detect

There were 104 unique external destination addresses and 2 unique internal destination addresses for a total of 106 destination addresses.

The table below lists the top five destinations.

Destination	Count
209.123.49.141	1353
66.135.208.201	294
212.112.162.203	256
192.151.53.10	185
66.135.192.227	150

Description of detect

The Snort documentation describes this detect as follows:

> "It's a part of the http preprocessor. Basically, if the http decoding routine finds a %00 in an http request, it will alert with this message. Sometimes you may see false positives with sites that use cookies with urlencoded binary data, or if you're scanning port 443 and picking up SSLencrypted traffic . If you're logging alerted packets you can check the actual string that caused the alert. Also, the unicode alert is subject to the same false positives with cookies and SSL. Having the packet dumps is the only way to tell for sure if you have a real attack on your hands, but this is true for any content-based alert."

In the universities logs all of the top 5 destinations are external web servers and the top 5 sources are internal machines. Someone internal may be attempting a web attack but without the packet data it's unknown if these are real or false positives. There are a few alerts that are from external sources to internal web servers but again without the packet data it's unknown if these are real or false positives.

Defensive Recommendations

Investigate the packet data for the alerts and as always be sure all web servers have the latest patches.

Correlation

Donald Gregory in his practical suggests adding code to CGI scripts to check inputs for “\0” before passing them to calls such as ‘open’.

Top Talkers

The table below lists the top ten talkers by number of alerts generated.

Source	Count	Alert(s)
212.179.116.236	14001	Watchlist
212.179.116.153	13398	Watchlist
MY.NET.198.221	7729	IRC
MY.NET.201.58	4289	Red worm
66.42.68.210	2787	Red worm
194.148.17.27	2713	SMB
MY.NET.235.110	2103	Tiny fragments, Red worm
212.179.27.6	1856	Watchlist
24.26.201.174	1694	Red worm
212.179.5.161	1608	Watchlist

212.179.116.236
---

inetnum: 212.179.100.0 - 212.179.124.255
netname: CABLES-CONNECTION
descr: CABLES-CUSTOMERS-CONNECTION
country: IL
admin-c: YK76-RIPE
tech-c: BHT2-RIPE
status: ASSIGNED PA
remarks: please send ABUSE complains to abuse@bezeqint.net
mnt-by: AS8551-MNT
mnt-lower: AS8551-MNT
notify: hostmaster@bezeqint.net
changed: hostmaster@bezeqint.net 20021029
source: RIPE

route: 212.179.96.0/19
descr: ISDN Net Ltd
origin: AS8551
notify: hostmaster@bezeqint.net
mnt-by: AS8551-MNT
changed: hostmaster@bezeqint.net 20030505
source: RIPE

role: BEZEQINT HOSTMASTERS TEAM
address: bezeq-international
address: 40 hashacham
address: petach tikva 49170 Israel
phone: +972 1 800800110
fax-no: +972 3 9203033

e-mail: hostmaster@bezeqint.net
admin-c: YK76-RIPE
tech-c: MR916-RIPE
nic-hdl: BHT2-RIPE
remarks: Please Send Spam and Abuse ONLY to abuse@bezeqint.net
mnt-by: AS8551-MNT
changed: hostmaster@bezeqint.net 20021029
changed: hostmaster@bezeqint.net 20030204
source: RIPE

person: Yuval Keinan
address: bezeq-international
address: 40 hashacham
address: petach tikva 49170 Israel
phone: +972 1 800800110
fax-no: +972 3 9203033
e-mail: hostmaster@bezeqint.net
mnt-by: AS8551-MNT
nic-hdl: YK76-RIPE
changed: hostmaster@bezeqint.net 20021215
changed: hostmaster@bezeqint.net 20030204
source: RIPE

66.42.68.210

OrgName: Pac-West Telecomm, INC.
OrgID: PWTI
Address: 1776 W. March Lane
Address: Suite 250
City: Stockton
StateProv: CA
PostalCode: 95207
Country: US

NetRange: 66.42.0.0 - 66.42.127.255
CIDR: 66.42.0.0/17
NetName: MDSG-PACWEST-1BLK
NetHandle: NET-66-42-0-0-1
Parent: NET-66-0-0-0-0
NetType: Direct Allocation
NameServer: NS1.MDSG-PACWEST.COM
NameServer: NS2.MDSG-PACWEST.COM
NameServer: NS3.MDSG-PACWEST.COM
NameServer: NS4.MDSG-PACWEST.COM
NameServer: NS5.MDSG-PACWEST.COM
NameServer: NS6.MDSG-PACWEST.COM
Comment: ADDRESSES WITHIN THIS BLOCK ARE NON-PORTABLE
RegDate: 2000-11-10
Updated: 2002-11-15

TechHandle: ZP86-ARIN
TechName: Administrator
TechPhone: +1-800-722-9378
TechEmail: admin@mdsg-pacwest.com

OrgTechHandle: ZP86-ARIN
OrgTechName: Administrator
OrgTechPhone: +1-800-722-9378
OrgTechEmail: admin@mdsg-pacwest.com

194.148.17.27
---

inetnum: 194.148.17.0 - 194.148.17.63
netname: FIVB-NET
descr: Federation International de Volleyball
descr: Avenue de la Gare 12
descr: 1001 Lausanne
country: CH
admin-c: MC1375-RIPE
tech-c: MC1375-RIPE
status: ASSIGNED PA
mnt-by: AS3334-MNT
changed: maintenance@pingnet.ch 20000824
source: RIPE

route: 194.148.0.0/16
descr: PINGNET-C1-C256
origin: AS6756
mnt-by: AS6756-MNT
changed: mario.cantieni@tiscali.ch 20010713
source: RIPE

person: Mario Cantieni
address: Tiscali DataComm AG
address: Herostrasse 9
address: CH-8048 Zurich
address: Switzerland
phone: +41 1 434 70 21
fax-no: +41 1 434 70 20
e-mail: cantieni@pingnet.ch
nic-hdl: MC1375-RIPE
mnt-by: AS3334-MNT
changed: cantieni@pingnet.ch 20020506
source: RIPE

212.179.27.6
---

inetnum: 212.179.27.4 - 212.179.27.7
netname: ADI-ASSOCIATION
descr: ADI-ASSOCIATION-SERIAL
country: IL
admin-c: NP469-RIPE
tech-c: NP469-RIPE
status: ASSIGNED PA
notify: hostmaster@isdn.net.il
mnt-by: RIPE-NCC-NONE-MNT
changed: hostmaster@isdn.net.il 20000106
source: RIPE

route: 212.179.0.0/18
descr: ISDN Net Ltd.
origin: AS8551
notify: hostmaster@bezeqint.net
mnt-by: AS8551-MNT
changed: hostmaster@bezeqint.net 20020618
source: RIPE

person: Nati Pinko
address: Bezeq International
address: 40 Hashacham St.
address: Petach Tikvah Israel
phone: +972 3 9257761
e-mail: hostmaster@isdn.net.il
nic-hdl: NP469-RIPE
changed: registrar@ns.il 19990902
source: RIPE

24.26.201.174

OrgName: Road Runner
OrgID: RRMA
Address: 13241 Woodland Park Road
City: Herndon
StateProv: VA
PostalCode: 20171
Country: US

NetRange: 24.24.0.0 - 24.29.255.255
CIDR: 24.24.0.0/14, 24.28.0.0/15
NetName: ROAD-RUNNER-1
NetHandle: NET-24-24-0-0-1
Parent: NET-24-0-0-0-0
NetType: Direct Allocation
NameServer: DNS1.RR.COM
NameServer: DNS2.RR.COM
NameServer: DNS3.RR.COM

NameServer: DNS4.RR.COM
Comment:
RegDate: 2000-06-09
Updated: 2002-08-22

TechHandle: ZS30-ARIN
TechName: ServiceCo LLC
TechPhone: +1-703-345-3416
TechEmail: abuse@rr.com

OrgTechHandle: IPTEC-ARIN
OrgTechName: IP Tech
OrgTechPhone: +1-703-345-3416
OrgTechEmail: abuse@rr.com

OrgAbuseHandle: ABUSE10-ARIN
OrgAbuseName: Abuse
OrgAbusePhone: +1-703-345-3416
OrgAbuseEmail: abuse@rr.com

212.179.5.161
---
inetnum: 212.179.5.160 - 212.179.5.191
netname: KIBBUTZ-GVAT-PLASTRO
mnt-by: INET-MGR
descr: KIBBUTZ-GVAT-PLASTRO-LAN
country: IL
admin-c: ZV140-RIPE
tech-c: MZ4647-RIPE
status: ASSIGNED PA
notify: hostmaster@isdn.net.il
changed: hostmaster@isdn.net.il 20020415
source: RIPE

route: 212.179.0.0/18
descr: ISDN Net Ltd.
origin: AS8551
notify: hostmaster@bezeqint.net
mnt-by: AS8551-MNT
changed: hostmaster@bezeqint.net 20020618
source: RIPE

person: Zehavit Vigder
address: bezeq-international
address: 40 hashacham
address: petach tikva 49170 Israel
phone: +972 1 800800110
fax-no: +972 3 9203033
e-mail: hostmaster@bezeqint.net

mnt-by: AS8551-MNT
nic-hdl: ZV140-RIPE
changed: hostmaster@bezeqint.net 20021027
changed: hostmaster@bezeqint.net 20030204
source: RIPE

person: Meron Ziv
address: Bezeq International
address: hashacham 40
address: petach tiqua
address: Israel
phone: +972-3-9257710
e-mail: hostmaster@bezeqint.net
nic-hdl: MZ4647-RIPE
changed: hostmaster@bezeqint.net 20010107
source: RIPE

Dshield didn't show any incidents attributed to the above IP addresses.

Portscan Analysis

There were 283624 UDP scans and 147062 TCP scans.

MY.NET did not appear in these logs like it did in the others. It looks as though 130.85.0.0 is the internal network.

The top ten sources performing UDP scans were all internal addresses.

Source	Count	Most Frequent Port Scanned
130.85.87.50	37494	27022
130.85.87.44	18548	27021
130.85.207.230	15484	6257
130.85.97.95	12787	7674
130.85.97.39	10469	22321
130.85.168.90	9834	22321
130.85.97.174	7561	1038
130.85.225.150	7450	1210
130.85.98.96	6943	22321
130.85.1.3	5017	57312

The top ten UDP ports scanned are shown in the following table.

Port	Protocol	Count
22321	dobol backdoor	52988
137	netbios	36843
27005	Gnutella	22731

7674	Imq SSL tunnel	21350
6257	WinMX	20006
43620	Unknown	10234
53	DNS	8125
43622	Unknown	4054
27020	HLTV	2105
14690	gamespy	1950

The top ten sources performing TCP scans were all external addresses.

Source	Count
146.164.34.42	12962
193.11.250.21	11322
213.84.229.115	10926
217.40.73.165	10374
217.70.4.246	9302
216.137.3.107	8837
152.1.193.6	6750
80.14.15.28	3009
62.177.176.163	2762
192.215.160.106	2655

The following table lists the top ten TCP ports scanned.

Port	Protocol	Count
443	SSL	25225
80	HTTP	20151
445	Microsoft-DS	18776
1433	Microsoft SQL server	16665
139	netbios	13373
0	Unknown	9434
21	FTP	7063
135	RPC Service DOS	3544
6346	Gnutella	2726
4000	ICQ	2603

The large number of dobol backdoor scans from internal addresses is one significant finding. These were being reported from many internal machines however so it could be the scan rule needs to be adjusted and these are false positives. If not the university needs to check the machines in question. Among these are 130.85.98.96, 130.85.97.212, 130.85.240.78, and 130.85.97.86. Also there appears to be a lot of internal use of file sharing and game playing sites.

The university may also want to consider adding some of the top external scanners to their watchlist.

Out-Of-Spec Analysis

There were 9673 OOS detects during the five day period.

Of these 6992 were logged because reserved bits in the TCP header were set. However with RFC3168 these bits are now used for Explicit Congestion Notification (ECN) and therefore are not out of spec.

Of the remaining 2585 detects the most prevalent were:

2048 packets with both the SYN and FIN flags were set.

128 packets where the TCP ack field was 0 and the SYN bit was not set.

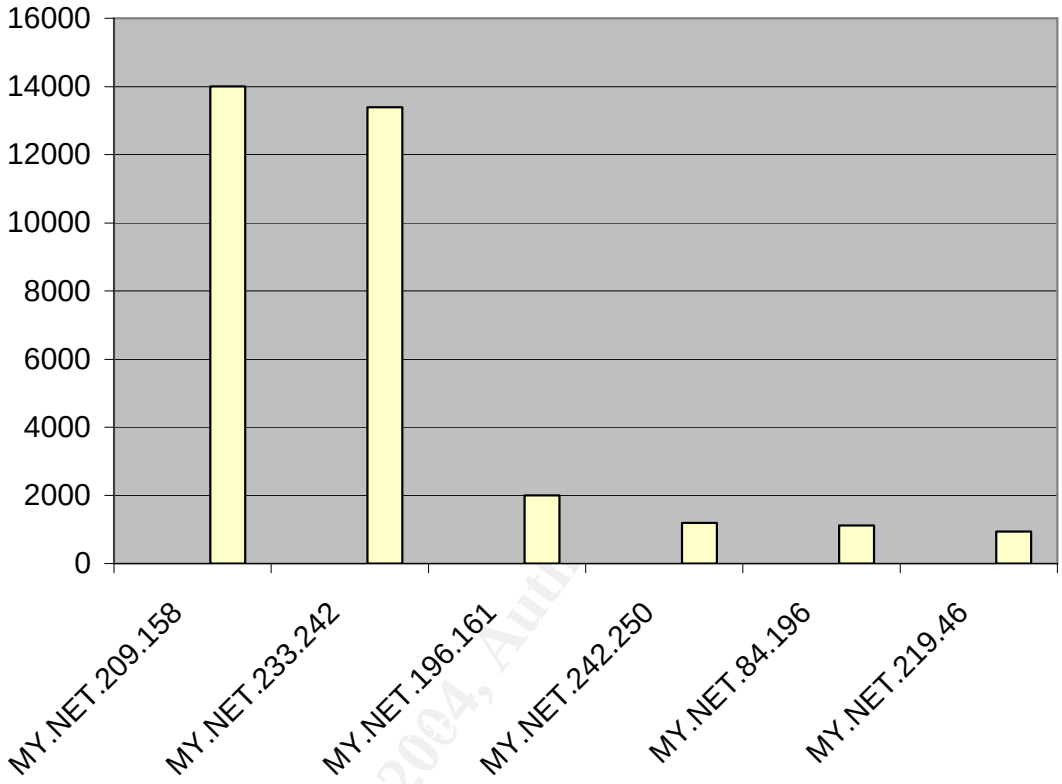
84 packets with no TCP flags set.

22 packets with both the don't fragment flag and the more fragments flag set.

Most of these are probably scans that are attempting to evade detection. The packets with both the don't fragment flag and the more fragments flag set are probably some type of attack that is trying to evade detection. It will have also generated an alert if the appropriate Snort rule is in place.

Link Graph

The following graph shows the machines most frequently accessed by IP addresses on the universities watchlist.



Security Recommendations

It is unknown where in the network the IDS is located. Whether to position it outside the border router and firewall or inside has long been debated. The advocates of placing it outside want to know what attacks are being attempted and those advocating inside are only concerned with what has made it through the perimeter defences. If in this case it is inside then the router and firewall need to be more restrictive. As mentioned earlier several of the top ten alerts could be blocked by router ACL's or a firewall. There is probably no need to allow netbios traffic to enter from the outside for example. Also machines should be scanned for the Red worm / Adore worm. And of course all machines should have the latest patches applied.

Description of Analysis Process

To analyse the alert data I first combined all five days worth of alerts into a single file. Also as several others mentioned in their practical I then removed the scan data from the alert files since there were separate scan files. I then used the Perl script included in Les Gordon's practical. This script extracted the following components from the alerts logs:

- Alert name
- Number of alerts
- Number of external sources
- Number of external destinations
- Number of internal sources
- Number of internal destinations
- Number of inbound alerts
- Number of outbound alerts
- Number of internal to internal alerts
- Number of external to external alerts

The script also generated a list of the source addresses, destination address, source ports, etc ordered by the number of occurrences. This allowed me to easily find the top ten attacks.

After getting the above information across all the alerts I then used grep to extract each of the top ten alerts into a separate file and ran the script on that file. This provided information such as the top source and destination addresses for each attack type.

I then modified the script slightly to use it to generate similar information for the scan and OOS logs.

These tools allowed me to generate the tables required for the analysis. I then used google searches to determine the nature of the attacks. In some cases to gain additional information on an attack I also used grep, wc, and sort.

References

Anonymous. "Port Numbers." Assigned Numbers. Nov. 13 2002
URL: http://www.iana.org/assignments/port-numbers (Nov. 15 2002).

Anonymous. “Firewall Forensics”
URL: http://www.first.org/events/progconf/2000/D1-03.pdf

Anonymous. “Buffer Overflows With Content”
URL: http://www.cccure.org/amazon/idssignature.pdf

Anonymous. "Ethernet Vendor Address Assignments."
URL: http://www.lex-con.com/protocols/en-addr.txt
URL: http://coffer.com/mac_find/

Anonymous. "Snort fragmentation rules”
URL: http://www.snort.org/docs/writing_rules/chap2.html#tth_sEc2.3.7

Anonymous. "Snort RPC rules”
URL: http://www.snort.org/docs/writing_rules/chap2.html#tth_sEc2.3.21

Anonymous. “The Code Red Worm”
URL: http://www.ciac.org/ciac/bulletins/l-117.shtml

Anonymous. “Reverse DNS Lookup”
URL: http://remote.12dt.com/rns/
URL: http://www.dnsstuff.com/

Anonymous. “Babel Fish Translation”
URL: http://world.altavista.com/

Anonymous. “Vulnerabilities in PCNFSD”
URL: http://www.cert.org/advisories/CA-1996-08.html

Anonymous. “Buffer Overflow Vulnerability in mountd”
URL: http://www.cert.org/advisories/CA-1998-12.html

Anonymous. “pcserver”
URL: http://www.doc.ic.ac.uk/~mac/manuals/hpux-manual-pages/hpux/usr/man/man1m/pcserver.1m.html

Anonymous. “Bugs, Holes, Patches”
URL: http://www.nipc.gov/cybernotes/2001/cyberissue2001-07.pdf

Anonymous. "Cybercops”
URL: http://www.computercops.biz/modules.php?name=nmap

Anonymous. " Trojanlistan”
URL: http://www.simovits.com/sve/nyhetsarkiv/1999/nyheter9902.html

Anonymous. "trojans"
URL: http://www.tigertools.net/trojans.txt

Anonymous. "Backdoor Found"
URL: http://www.saintcorporation.com/demo/saint_tutorials/backdoor_found.html

Anonymous. " Troj/Muska52-13"
URL: http://www.sophos.com/virusinfo/analyses/trojmuska5213.html

Anonymous. " Backdoor.Skun"
URL: http://securityresponse.symantec.com/avcenter/venc/data/backdoor.skun.html

Anonymous. "ADM Worm"
URL: http://www.redhat.com/archives/linux-security/1999- March/msg00004.html

Anonymous. "The Lion Internet Worm DDOS Risk"
URL: http://ciac.llnl.gov/ciac/bulletins/I-064.shtml

Anonymous. "MscanWorm"
URL: http://www.simovits.com/trojans/tr_data/y2169.html

Anonymous. " ProMail trojan"
URL: http://www.psc.ru/sergey/bgtraq/STRANGEVIRUS/TROYAN/promail.htm

Anonymous. " DShield"
URL: http://www.dshield.org/

Anonymous. " NetWatchman"
URL: http://www.mynetwatchman.com/

Anonymous. " fragrouter"
URL: http://packetstorm.widexs.nl/UNIX/IDS/nidsbench/fragrouter.html

Aaron Hackworth. "GCIA Practical"
URL: http://www.giac.org/practical/GCIA/Aaron_Hackworth_GCIA.pdf

Brad Doctor and Martin Markgraf. "httpd and sunrpc probes"
URL: http://lists.jammed.com/incidents/2001/05/0100.html

Bryce Alexander. "Port 137 Scan"
URL: http://www.sans.org/resources/idfaq/port_137.php

Donald Cunningham. "GCIA Practical."
URL: http://www.giac.org/practical/GCIA/Donald_Cunningham_GCIA.pdf

Donald Gregory. "GCIA Practical"
URL: http://www.giac.org/practical/GCIA/Donald_Gregory_GCIA.pdf

Donald Merchant. " GCIA Practical"

URL: http://www.giac.org/practical/Donald_Merchant_GCIA.doc

Doug Kite. "GCIA Practical"
URL: http://www.giac.org/practical/GCIA/Doug_Kite_GCIA.pdf

Juergen P. Meier, Chris Brenton. "pcserver attack"
URL: http://www.shmoo.com/mail/firewalls/mar00/msg00020.shtml

K. Ramakrishnan. "RFC3168"
URL: http://www.rfc-editor.org/rfc/rfc3168.txt

Les Gordon. "GCIA Practical"
URL: http://www.giac.org/practical/GCIA/Les_Gordon_GCIA.doc

Mark Donaldson. " RPC Detect"
URL: http://cert.uni-stuttgart.de/archive/intrusions/2003/01/msg00209.html

Peter Szczepankiewicz. "Illegal Fragmentation"
URL: http://cert.uni-stuttgart.de/archive/intrusions/2002/11/msg00019.html

Robert Buckley. "Illegal fragmentation"
URL: http://archives.neohapsis.com/archives/incidents/2002-04/0054.html

Roger Thompson. "Illegal Fragmentation"
URL: http://cert.uni-stuttgart.de/archive/intrusions/2002/08/msg00239.html

Sami Rautiainen. "Adore Worm"
URL: http://www.f-secure.com/v-descs/adore.shtml

Scott Gregory. "Illegal Fragmentation"
URL: http://cert.uni-stuttgart.de/archive/intrusions/2002/08/msg00106.html

Sebastien Pratte. "GCIA Practical"
URL: http://www.giac.org/practical/GCIA/Sebastien_Pratte_GCIA.pdf

Thomas H Ptacek. " Eluding Network Intrusion Detection"
URL: http://secinf.net/info/ids/idspaper/idspaper.html

Wietse Venema. " Secure Portmapper"
URL: http://ftp.porcupine.org/pub/security/