Divert Sockets mini−HOWTO

Table of Contents

Divert Sockets mini−HOWTO

Ilia Baldine, `ibaldin@anr.mcnc.org`

v1.1, 27 February 2000

This document describes how to get, compile and use FreeBSD divert sockets under Linux 2.2.12.

7.Advanced issues

8.Geting More Information

9.Future work

1.Copyright

2.Disclaimer

This work has been done as part of a DARPA−funded network security project. Neither I (Ilia Baldine), nor my employer (MCNC) nor DARPA can be held accountable for any damage real or potential that can come to you through the use by you or other parties of the code and/or procedures described in this document. As many other network mechanisms, divert sockets can be used as much for evil as for good and its *your* choice!

3.Foreword

```
Here's an easy game to play,
Here's an easy thing to say:

If a packet hits a pocket
 on a socket on a port
And the bus is interrupted
  as a very last resort,
And the address of the memory
```

```
  makes your floppy disk abort,
Then the socket packet pocket
  has an error to report!!

If your cursor finds a menu item
  followed by a dash,
And the double clicking icon puts  your
  window in the trash,
And your data is corrupted 'cause the
  index doesn't hash,
Then the situation's hopeless, and your
  system's gonna crash!

YOU CAN'T SAY THIS? WHAT A SHAME SIR!
WE'LL FIND ANOTHER GAME SIR

If the label on the cable on the table
  at your house,
Says the network is connected to
  the button on your mouse,
But your packets want to tunnel
  on another protocol,
That's repeatedly rejected
  by the printer down the hall,
And your screen is all distorted
  by the side effects of gauss
So your icons in the window are
  as wavy as a souse,
Then you  may as well reboot and
  go out with a bang,
'Cause as sure as I'm a poet,
  the sucker's gonna hang!
When the copy of your floppy's
  getting sloppy on the disk
And the microcode instructions cause
 unnecessary risc,
Then you have to flash your memory and
  you'll want to RAM your  ROM
Quickly turn off your computer and
  be sure  to tell your mom!

-- Anonymous
```

4.Introduction

Ever wish you could intercept packets traveling up or down the IP stack of your host? And I'm not talking about listening in, like raw sockets or libpcap (tcpdump). I mean literally stop the packet from further propagating through the IP stack and then (possibly after some changes), reinjecting it back? Well, the time to dream is over, because divert sockets for Linux are here!

Divert sockets do exactly that − they filter out certain packets based on firewall specifications and bring them to you in user space. You then have the freedom of simply reinjecting them back as if nothing happened, mangling them first and then reinjecting them, or not reinjecting them at all.

As the name suggests, this mechanism utilizes a special type of RAW socket called divert (IPPROTO_DIVERT) that allow you to *receive* and *send* on them just like regular sockets. The difference is that a divert socket is bound to a port, into which the firewall can be instructed to send certain packets. Anything that a firewall can filter out can be sent into a divert socket.

Divert sockets first appeared as part of FreeBSD. Divert sockets under Linux is a port of this mechanism that strives to be source−code compatible in terms of user−space programs that utilize it.

5.Getting and Compiling the Source Code

In order to use divert sockets under Linux you will need two things − the kernel source code that has been patched for divert sockets and the source code to ipchains−1.3.9 that, also, has been patched to use divert sockets.

5.1 Getting *The Source*

Both pieces of source code can be retrieved from the divert socket web−site http://www.anr.mcnc.org/~divert You can get the source code for divert sockets kernel in two forms − as a patch to linux−2.2.12 that you have to apply to a fresh 2.2.12 source, or as an already patched kernel tarball (much larger than the patch). *ipchains* source is provided as complete source tarball only.

5.2 Compiling

Compiling *ipchains* is straightforward − simply say

```
make
```

in the ipchains−1.3.9 subdirectory.

When compiling the divert−socket kernel − use your favorite way of configuring it:

```
make config
or

make menuconfig
or

make xconfig
```

Don't forget to enable "Prompt for development and/or incomplete code/drivers" before proceeding. There are only three compile−time options that affect the behavior of divert sockets and they are explained in the following section

Kernel compile−time options

In order to enable divert sockets in your kernel you must enable firewalling and IP firewalling first. The three kernel compile−time options that affect the behavior of divert sockets are:

IP: divert sockets

Enables the divert sockets in your kernel.

IP: divert pass−through

Changes the behavior of DIVERT rules: by default if a DIVERT rule is present in a firewall and no application is listening on the port that the rule specifies, any packet that satisfies the rule is silently dropped, as if it were a DENY rule.
Enabling the pass−through mode results in such packets continuing their way through the IP stack as if nothing happened. This could be helpful if you want to have a static rule in the firewall, but don't always want to listen on it.

IP: always defragment

Changes the way that the sockets deal with fragmentation. By default the divert socket receives individual fragments of packets that are larger than MTU, which it then forwards to user space. The burden of defragmentation in this case lies with the application listening on the divert socket. Also, an application cannot inject any fragments that are larger than MTU, because they will be dropped (this is the limitation of the kernel, not the divert sockets − Linux kernels up to 2.2.x do NOT fragment raw packets with IP_HDRINCL option set). Typically, thats OK, since if you simply reinject the fragments the way you received them, everything will work fine, since none of them are going to be larger than MTU.
If you enable the *always defragment* option, then all the defragmentation will be done for you in the kernel. This severely affects the performance of the interception mechanism, since now every large packet you want intercepted will first have to be reassembled prior to being forwarded to you, and then, if you choose to reinject it − it will have to be fragmented again (the kernel with this option will be enabled to fragment raw packets with IP_HDRINCL)
This was the only option available for divert sockets under Linux 2.0.36 because of the way the firewall code was structured − it only looked at the first fragment of every packet and passed all other fragments without looking at them. This way, if the first fragment were dropped by the firewall, the rest of them would be eventually discarded by the defragmenter. That's why in order for DIVERT sockets to work you were forced to compile the *always defragment* option in, so that you would always get the whole packet diverted to you and not just the first fragment.
In 2.2.12, thanks to changes in the firewall code you now have an option of having the kernel or yourself doing fragmentation/defragmentation.
NOTE: the defragmentation feature has not been added as of release 1.0.4 of divert sockets. It is in the works though.

6.Using Divert Sockets

This section will give you examples of how divert sockets can be used and how they are different of other packet interception mechanisms out there.

6.1 Divert sockets vs. other stuff

There are other mechanisms out there that have similar functionality. Here is why they are different:

Netlink sockets

Netlink sockets can intercept packets just like divert sockets by using firewall filter. They have a special type (AF_NETLINK) and on the surface seem to do the same thing. Two major differences are:

- Netlink sockets have no ports, so it is difficult to have multiple processes intercepting different things (divert sockets have a standard 16−bit port space, which means you can have 65535 processes diverting packets independently)
- Netlink sockets have no easy way of injecting the packets that are outbound (going on the wire) because no special precautions are taken not to reintercept the same packet over and over again as it is injected. Divert sockets do this automatically

To be fair, the scope of netlink sockets is wider than this. In general, netlink mechanism is intended to allow communication between kernel and user space. There are, for instance, netlink routing sockets that allow you to communicate with the routing subsystem. However, as a packet interception mechanism, they are not as robust as divert sockets.

Raw sockets

RAW sockets can be a good way to listen in on traffic (especially under Linux, where RAW sockets can listen in on TCP and UDP traffic, although most other UNI*s do not allow that) but a RAW socket can't stop a packet from propagating through the IP stack − it simply gives you a copy of the packet and there is no way to inject it inbound (on the way up the stack) − only outbound. Also, you can only filter pockets out by the protocol number, which you specify when you open a RAW socket. There is no link between the firewall and RAW sockets.

libpcap

More commonly known for the tool it facilitates − tcpdump, libpcap lets you listen in on traffic that hits your interface (whether it be ppp or eth or whatever). For ethernet it can also put your NIC into a promiscuous mode, so that it will forward to IP the traffic that not only is link−layer addressed to it, but to others on the same segment. Of course, libpcap allows for no way of actually stopping packets from propagating and no way to inject. In fact, libpcap is in many ways orthogonal to divert sockets.

6.2 Discussion on firewall chains

Linux provides you with three default chains: input, output and forward. There are also accounting chains, but they are of no consequence here. Depending on the packet origin it traverses one or more of these chains:

Input chain

is traversed by all packets that come into the host − packets that are addressed to it and packets that will be forwarded by it.

Output chain

is traversed by all packets originating in the host and by all forwarded packets

Forward chain

is traversed only by the forwarded packets.

The order in which a forwarded packet traverses the chains is:

1. Input
2. Forward
3. Output

This may sometimes create problems for the interception if you are interested in a certain type of packets that may or may not originate on your host. A lot of times it is not clear which chain to use.

As a rule of thumb, forward chain should only be used to filter packets that are forwarded and are not originating and are not addressed to your host. If you are interested in a combination of both forwarded packets and packets that are originating or addressed to your host, then use input or output chain instead. Intercepting on forward and input or output chain for the same type of packet at the same time will create problems in reinjection and, more importantly, is unnecessary.

6.3 Using ipchains

The patched version of ipchains that you will need to retrieve from the website, is the tool that allows you to modify firewall rules from a shell (most people want that). It is also possible to set up firewall rules programmatically. See the example code for this − setting up a DIVERT rule would be similar to setting up a REDIRECT rule − specify DIVERT as a target and the divert port and you are set to go.

The ipchains syntax for setting up firewall rules remains the same. To specify a DIVERT rule you must specify `−j DIVERT <port num>` as a target, everything else remains the same. For instance

```
ipchains −A input −p ICMP −j DIVERT 1234
```

would set up a divert rule for ICMP packets to be diverted from input chain to a port 1234.

The following section explains how to use ipchains in conjunction with an interceptor user−space program.

6.4 Plain vanilla example

Example program

Here is an example program that reads packets from a divert socket, displays them and then reinjects them back. It requires that the divert port is specified on the command line.

```
#include <stdio.h>
#include <errno.h>
#include <limits.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <getopt.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <signal.h>

#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <net/if.h>
#include <sys/param.h>

#include <linux/types.h>
#include <linux/icmp.h>
#include <linux/ip_fw.h>

#define IPPROTO_DIVERT 254
#define BUFSIZE 65535

char *progname;

#ifdef FIREWALL

char *fw_policy="DIVERT";
char *fw_chain="output";
struct ip_fw fw;
struct ip_fwuser ipfu;
struct ip_fwchange ipfc;
int fw_sock;

/* remove the firewall rule when exit */
void intHandler (int signo) {

  if (setsockopt(fw_sock, IPPROTO_IP, IP_FW_DELETE, &ipfc, sizeof(ipfc))==-1) {
    fprintf(stderr, "%s: could not remove rule: %s\n", progname, strerror(errno));
    exit(2);
```

```
  }

  close(fw_sock);
  exit(0);
}

#endif

int main(int argc, char** argv) {
  int fd, rawfd, fdfw, ret, n;
  int on=1;
  struct sockaddr_in bindPort, sin;
  int sinlen;
  struct iphdr *hdr;
  unsigned char packet[BUFSIZE];
  struct in_addr addr;
  int i, direction;
  struct ip_mreq mreq;

  if (argc!=2) {
    fprintf(stderr, "Usage: %s <port number>\n", argv[0]);
    exit(1);
  }
  progname=argv[0];

  fprintf(stderr,"%s:Creating a socket\n",argv[0]);
  /* open a divert socket */
  fd=socket(AF_INET, SOCK_RAW, IPPROTO_DIVERT);

  if (fd==-1) {
    fprintf(stderr,"%s:We could not open a divert socket\n",argv[0]);
    exit(1);
  }

  bindPort.sin_family=AF_INET;
  bindPort.sin_port=htons(atol(argv[1]));
  bindPort.sin_addr.s_addr=0;

  fprintf(stderr,"%s:Binding a socket\n",argv[0]);
  ret=bind(fd, &bindPort, sizeof(struct sockaddr_in));

  if (ret!=0) {
    close(fd);
    fprintf(stderr, "%s: Error bind(): %s",argv[0],strerror(ret));
    exit(2);
  }
#ifdef FIREWALL
  /* fill in the rule first */
  bzero(&fw, sizeof (struct ip_fw));
  fw.fw_proto=1; /* ICMP */
  fw.fw_redirpt=htons(bindPort.sin_port);
  fw.fw_spts[1]=0xffff;
  fw.fw_dpts[1]=0xffff;
  fw.fw_outputsize=0xffff;

  /* fill in the fwuser structure */
  ipfu.ipfw=fw;
  memcpy(ipfu.label, fw_policy, strlen(fw_policy));

  /* fill in the fwchange structure */
  ipfc.fwc_rule=ipfu;
  memcpy(ipfc.fwc_label, fw_chain, strlen(fw_chain));
```

```
  /* open a socket */
  if ((fw_sock=socket(AF_INET, SOCK_RAW, IPPROTO_RAW))==-1) {
    fprintf(stderr, "%s: could not create a raw socket: %s\n", argv[0], strerror(errno));
    exit(2);
  }

  /* write a rule into it */
  if (setsockopt(fw_sock, IPPROTO_IP, IP_FW_APPEND, &ipfc, sizeof(ipfc))==-1) {
    fprintf(stderr, "%s could not set rule: %s\n", argv[0], strerror(errno));
    exit(2);
  }

  /* install signal handler to delete the rule */
  signal(SIGINT, intHandler);
#endif /* FIREWALL */

  printf("%s: Waiting for data...\n",argv[0]);
  /* read data in */
  sinlen=sizeof(struct sockaddr_in);
  while(1) {
    n=recvfrom(fd, packet, BUFSIZE, 0, &sin, &sinlen);
    hdr=(struct iphdr*)packet;

    printf("%s: The packet looks like this:\n",argv[0]);
        for( i=0; i<40; i++) {
                printf("%02x ", (int)*(packet+i));
                if (!((i+1)%16)) printf("\n");
        };
    printf("\n");

    addr.s_addr=hdr->saddr;
    printf("%s: Source address: %s\n",argv[0], inet_ntoa(addr));
    addr.s_addr=hdr->daddr;
    printf("%s: Destination address: %s\n", argv[0], inet_ntoa(addr));
    printf("%s: Receiving IF address: %s\n", argv[0], inet_ntoa(sin.sin_addr));
    printf("%s: Protocol number: %i\n", argv[0], hdr->protocol);

    /* reinjection */

#ifdef MULTICAST
   if (IN_MULTICAST((ntohl(hdr->daddr)))) {
        printf("%s: Multicast address!\n", argv[0]);
        addr.s_addr = hdr->saddr;
        errno = 0;
        if (sin.sin_addr.s_addr == 0)
            printf("%s: set_interface returns %i with errno =%i\n", argv[0], setsockopt(fd, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr)), errno);
    }
#endif

#ifdef REINJECT
   printf("%s Reinjecting DIVERT %i bytes\n", argv[0], n);
   n=sendto(fd, packet, n ,0, &sin, sinlen);
   printf("%s: %i bytes reinjected.\n", argv[0], n);

   if (n<=0)
     printf("%s: Oops: errno = %i\n", argv[0], errno);
   if (errno == EBADRQC)
     printf("errno == EBADRQC\n");
   if (errno == ENETUNREACH)
     printf("errno == ENETUNREACH\n");
#endif
```

```
    }
}
```

You can simply cut−n−paste the code and compile it with your favorite compiler. If you want to enable reinjection − compile it with the −DREINJECT flag, otherwise it will only do the interception.

In order to get it to work, compile the kernel and ipchains−1.3.8 as described above. Insert a rule into any of the firewall chains: input, output or forward, then send the packets that would match the rule and watch them as they fly through the screen − your interceptor program will display them and then reinject them back, if appropriately compiled.

For example:

```
ipchains -A output -p TCP -s 172.16.128.10 -j DIVERT 4321
interceptor 4321
```

will divert and display all TCP packets originating on host 172.16.128.10 (for instance if your host is a gateway). It will intercept them on the output just before they go on the wire.

If you did not compile the pass through option into the kernel, then inserting the rule effectively will create a DENY rule in the firewall for the packets you specified until you start the interceptor program. See more on that above

If you want to set a firewall rule through your program, compile it with −DFIREWALL option and it will divert all ICMP packets from the output chain. It will also remove the DIVERT rule from the firewall when you use Ctrl−C to exit the program. In this case using pass−through vs. non−pass−through divert sockets makes virtually no difference.

6.5 The sky's the limit

As far as what you can use divert sockets for − your imagination would be the limiting factor. I would be interested to hear about applications that utilize divert sockets.

So, have fun!

7.Advanced issues

7.1 Packet Mangling

After you intercept a packet, it is possible to change its header or contents before reinjecting it back. Here are a few rules you might need to keep in mind:

- IP header checksum is always recalculated on injection
- IP ID field is filled in for you if you leave it 0.
- The length of the packet is updated for you.

All other parts of the IP header can be modified and its up to you to insure their sanity.

7.2 Injection with no interception

It is not necessary to intercept a packet in order to inject it. You can form your own packets and inject them into an open and bound divert socket. The header rules from above apply.

In addition, you need to pass to the divert socket a `sockaddr_in` structure (see example program), which will tell the socket where to inject. If you leave the structure 0−ed out or pass a NULL − the divert socket will attempt to inject the packet in the outbound direction (on the wire). If instead you fill the `sockaddr_in` structure with the address of one of the local interfaces, the divert socket will attempt to inject the packet inbound, as if it came from that interface. All addresses, of course, should be in network byte order.

Injection of packets that look like they are being forwarded by your host must include an address of the incoming interface (actually − any valid interface address will probably work).

7.3 Fragmentation

As of this reading, the divert sockets do not handle the defragmentation and fragmentation of diverted packets − you always get the fragments as they are on the wire and you should not inject fragments larger than PMTU. It is anticipated that the fragmentation/defragmentation capability will be added in the near future.

8.Geting More Information

8.1 The website

As mentioned above, most of the information about divert sockets can be found on the Divert Sockets for Linux website http://www.anr.mcnc.org/~divert.

8.2 The mailing list

There is also a mailing list, whose archive can be found at the website. To join the mailing list send email with an empty subject and the following line in the body:

```
subscribe divert
```

to anr−majordomo@list.anr.mcnc.org. The list address is divert@list.anr.mcnc.org.

To unsubscribe, send mail to anr−majordomo@list.anr.mcnc.org with an empty subject and the following line in the body:

```
unsubscribe divert
```

9.Future work

As mentioned in the disclaimer, work on divert sockets is done as part of a DARPA−funded network security effort. We will continue to port divert sockets to further versions of the kernel as time permits. Given that 2.4 kernel is on the horizon, in all likelihood we will skip 2.3.x series altogether.
